(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A Porting Methodology for Parallel Database Systems

by

Stanley Hugh Watkins

September 1993

Thesis Advisor: David K Hsiao

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School — 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Porting Methodology for Parallel Database Systems (U)

12. PERSONAL AUTHOR(S)
Watkins, Stanley Hugh

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 12/92 TO 09/93	September 1993	97

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel Database, Multilingual and Multimodal Database, MultiBackend Database Computer, Porting, MDBS, Heterogenous Database System

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Multibackend Database Supercomputer (MDBS) pioneered in the Naval Postgraduate School Laboratory for Database Systems Research offers an elegant solution to the four most pressing problems associated with the traditional approach to very large database management systems: capacity growth, performance improvement, data sharing, and resource consolidation. The purpose of this thesis is to develop a theory of system software portability for this large and complex network application which will facilitate others in the installation and utilization of MDBS.

The first challenge is the almost total lack of documentation about MDBS software of use to system porters. A second set of issues revolves around the use of hardware by MDBS, particularly the use of mass storage devices for the storage and manipulation of base- and meta-data. A third challenge concerns the portability of system calls, shell programs, and the C language implementation. A final set of portability issues arises from the extensive use of inter-process and inter-machine communications by MDBS.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. David K. Hsiao	22b. TELEPHONE (Include Area Code) (408) 656-2253	22c. OFFICE SYMBOL CS/Hq

Our approach to this project involves first analyzing the aforementioned portability issues. This analysis is tested by porting the most advanced version of MDBS software to a different platform supported by different hardware and operating system software.

This thesis provides a framework in which to understand and assess specific portability concerns about MDBS. We describe the original routines for accessing the mass storage devices and explain why it was necessary to modify them for portability. We identify and discuss other hardware-specific information coded into MDBS. We identify and correct problems related to the recompilation of the MDBS code on the new platform. We provide a detailed analysis of the requirements for and the implementation of inter-process and inter-machine communications for MDBS. In addition, we expand system debugging features, improve documentation, provide a new demonstration database, and offer advice for future porters of MDBS.

A Porting Methodology for Parallel Database Systems

by
Stanley Hugh Watkins
Major, United States Marine Corps
Bachelor of Science (Political Science), United States Naval Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author: [illegible]
Stanley Hugh Watkins

Approved By: [illegible]
Dr. David K. Hsiao, Thesis Advisor

[illegible]
Dr. C. Thomas Wu, Second Reader

[illegible]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The Multibackend Database Supercomputer (MDBS) pioneered in the Naval Postgraduate School Laboratory for Database Systems Research offers an elegant solution to the four most pressing problems associated with the traditional approach to very large database management systems: capacity growth, performance improvement, data sharing, and resource consolidation. The purpose of this thesis is to develop a theory of system software portability for this large and complex network application which will facilitate others in the installation and utilization of MDBS.

The first challenge is the almost total lack of documentation about MDBS software of use to system porters. A second set of issues revolves around the use of hardware by MDBS, particularly the use of mass storage devices for the storage and manipulation of base- and meta-data. A third challenge concerns the portability of system calls, shell programs, and the C language implementation. A final set of portability issues arises from the extensive use of inter-process and inter-machine communications by MDBS.

Our approach to this project involves first analyzing the aforementioned portability issues. This analysis is tested by porting the most advanced version of MDBS software to a different platform supported by different hardware and operating system software.

This thesis provides a framework in which to understand and assess specific portability concerns about MDBS. We describe the original routines for accessing the mass storage devices and explain why it was necessary to modify them for portability. We identify and discuss other hardware-specific information coded into MDBS. We identify and correct problems related to the recompilation of the MDBS code on the new platform. We provide a detailed analysis of the requirements for and the implementation of inter-process and inter-machine communications for MDBS. In addition, we expand system debugging features, improve documentation, provide a new demonstration database, and offer advice for future porters of MDBS.

ACKNOWLEDGEMENTS

I would like to thank Dr. David K. Hsiao for his insight and advice in all database matters and for his assistance reviewing this thesis. Without his knowledge and expertise, the Multibackend Database System project and this thesis would not be possible.

I would also like to thank the support staff at the Naval Postgraduate School for their help. Especially, I would like to thank John Locke and Mike Williams for their assistance with MDBS software and hardware.

Finally, I would like to thank my wife, Stella, for her patience.

TABLE OF CONTENTS

LIST OF FIGURES

I. THE INTRODUCTION

A. THE BACKGROUND FOR THIS THESIS

Today, governments and corporations are struggling to come to grips with the explosive growth of information processing requirements. The Department of Defense alone spends over nine billion dollars annually and is highly dependent on general-purpose data processing hardware, software and services [United States House of Representatives, 1989, p. 1]. Database systems are an important part of the information processing challenge. Unfortunately, the performance of database systems have not kept pace with the technical advances in computer industry as a whole [Elmasri, 1989, p. 637].

The Office of Naval Research (ONR) has identified two interoperable information systems technology issues. The first of these is data sharing. Data sharing (or data exchange) involves the ability to transparently access other user's databases. Traditional heterogenous database management systems (dbms) do not support access by users employing other data models and languages. For example, IBM's relational dbms called DB2, cannot access non-relational databases. Instead, IBM replicates non-relational data in the heterogenous form in separate non-relational dbms. Any replication of data at different locations or in different data models is not an answer, since this introduces data integrity problems and increases storage requirements. The second technology issue identified by ONR is that of resource consolidation. Resource consolidation refers to more efficient use of database hardware, software, and support personnel by consolidating them into one computing environment. The proliferation of stand-alone (homogenous/heterogenous) dbms represents needless duplication in a time of shrinking budgets.

A third database technology issue is processing speed. General-purpose, von Neuman type computers are not optimized for the tasks performed upon databases. Large databases are maintained in the secondary storage (*e.g.*, disks). Advances in the speed of secondary-storage devices have not kept pace with advances in the speed of central processing units.

This results in processing delays on the ever-increasingly large amounts of information stored in the secondary storage [Elmasri, 1989, p. 638].

A fourth database technology issue is capacity growth. As organizations become larger and more sophisticated, their information processing needs grow. This is not just the problem of data storage. More significant is the need to process the increasing amounts of data. The traditional solution to this problem has been the acquisition of larger, more powerful machines. The restrictions on budget growth and the long lead-time associated with the acquisition of new hardware make this an impractical approach.

Therefore, a new approach is needed to deal with these four technology issues. Such alternatives are generically referred to as database machines [Elmasri, 1989, p. 638]. One database machine which elegantly addresses all four of these issues is the subject of a continuing project in the NPS Laboratory for Database Systems Research. The project focuses on the Multibackend Database Supercomputer (MDBS) which is designed to run on standard, off-the-shelf hardware and identical system software consisting of networked UNIX workstations. A single general-purpose workstation serves as a controller. Multiple, identical workstations with their respective high capacity drives for the storage of base and meta data serve as backends. The controller receives queries from the users and broadcasts them to the backends which then return the results to the controller for post processing and routing to the users.

Base data of any given database are clustered. The clustered data are placed across many backend disks. They are configured in a loosely-coupled, parallel architecture which leads to parallel accesses to the base data. MDBS is also scalable, since an increase in performance can be achieved by attaching additional parallel backends to share the workload. MDBS is unique in that the response-time reduction is in direct proportion to the number of backends employed [Hsiao, 1991, p. 44 and Hall, 1989]. Figure 1 illustrates the hardware architecture of MDBS. This is a highly scalable architecture whose performance can be tailored to the needs of the user.

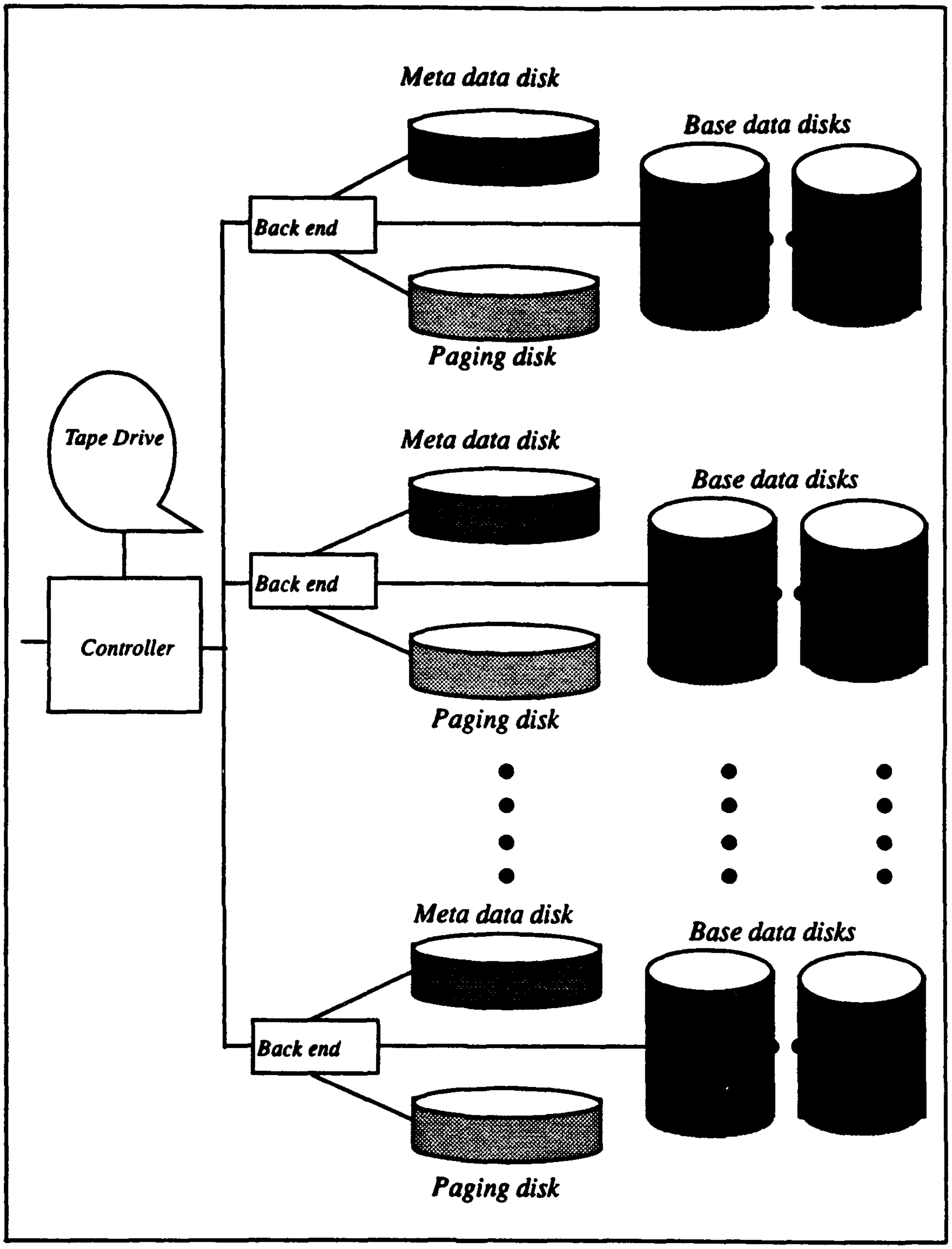

Figure 1. The Multibackend Database Supercomputer

The MDBS also answers ONR's concerns about data sharing among users employing different data models and data languages. MDBS employs a single "kernel" data model and data language. The capability to interface with users working in other data models and languages is provided for through the addition of language interface software modules with schema transformers and transaction translators [Hsiao and Kamel, 1989, p. 45]. This use of a multiple-data-models-and-languages-to-single-data- model-and-language ("many-to-one") mapping avoids the integrity pitfalls of data replication and permits global concurrency control. Language interfaces for the hierarchical, network, relational, and object-oriented models have been successfully integrated into MDBS. For more information on the construction of language interfaces, see [Bourgeois, 1993, p. 29]. A functional/ DAPLEX model interface is under construction. Most exciting is the cross-model accessing capability provided by the transformers and translators which allows, for instance, a relational user to access and perform operations upon a database created by a hierarchical user [Zawis, 1987, p. 30 - 74]. This cross-model access is transparent to the user, meaning that no retraining of the relational user in the hierarchical data manipulation language is required. Figure 2 represents the multimodel, multilingual, and cross-model accessing capabilities of MDBS. The Multibackend Database Supercomputer provides great flexibility in answering the problem of data sharing.

Finally, the Multibackend Database Supercomputer also addresses the problem of resource consolidation. The multi-model and multi-language capabilities of MDBS obsolete the plethora of stand-alone heterogenous database management systems currently in use in the DOD community. The performance of MDBS can be tailored to suit the needs of the user. A single database management system with a single scalable architecture, MDBS replaces many separate database management systems with different architectures. Support personnel only need to know and to support one system (*i.e.*, MDBS). The result is a desirable consolidation and standardization of resources.

Because it offers practical solutions to the problems of data sharing, resource consolidation, capacity growth and performance gain, we believe the Multibackend Database Supercomputer points the way to the future for users of very large heterogenous databases.

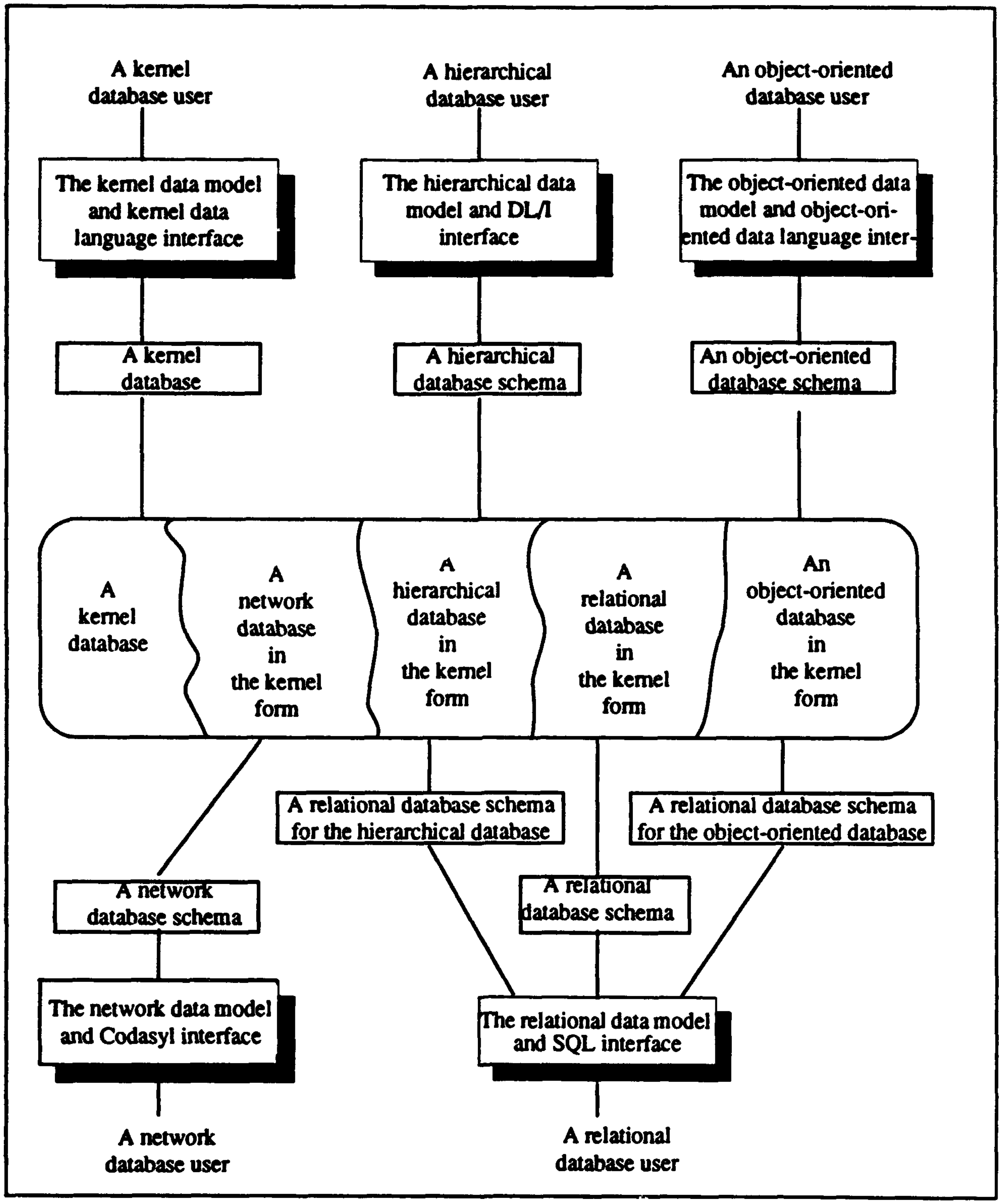
A kernel database user
A hierarchical database user
An object-oriented database user
The kernel data model and kernel data language interface
The hierarchical data model and DL/I interface
The object-oriented data model and object-oriented data language inter-
A kernel database
A hierarchical database schema
An object-oriented database schema
A kernel database
A network database in the kernel form
A hierarchical database in the kernel form
A relational database in the kernel form
An object-oriented database in the kernel form
A relational database schema for the hierarchical database
A relational database schema for the object-oriented database
A network database schema
A relational database schema
The network data model and Codasyl interface
The relational data model and SQL interface
A network database user
A relational database user

Figure 2. Multimodel, Multilingual, and Cross-Model Access Capabilities of MDBS

B. THE AIM OF THIS THESIS

As described above, MDBS is a very promising approach to the problems facing large database users. Only off-the-shelf hardware and identical system software are used. This supports a claim that MDBS is not hardware dependent and should be relatively easy to migrate to a new hardware platform. Indeed, an earlier version of MDBS was ported from its original platform consisting of one VAX-11/780 and two PDP-11/44 computers running VMS and RSX operating systems to its current platform consisting of seven ISI workstations running Berkeley UNIX operating system[Wong, 1986, p. 7-8]. The aim of this thesis is to migrate MDBS from its current platform to a newer risc-based platform running the SunOS operating system to develop a theory of system software portability for MDBS. This will become important as the interest in porting MDBS to other platforms grows. This theory of system software portability will address hardware issues, language issues, communication issues, and other issues important to any future porting of MDBS. It is hoped that this work will ease the job of moving MDBS to other platforms and spur more interest in MDBS.

C. THE SETTING OF THIS THESIS

The Multibackend Database Supercomputer is located in the Laboratory for Database Systems Research at the Naval Postgraduate School. The current hardware consists of seven ISI workstations based on the Motorola 68020 processor. All ISI workstations are using the Berkeley 4.3BSD operating system. The new hardware consists of a single Sun 4 Model 100 with two hard disk drives for a controller and two Sun 4 Model 280s each with three hard disk drives for backends. All three Sun machines operate under SunOS 4.1.1. The inter-machine-communications link remains a standard ethernet cable. For a detailed description of the old and new platforms, see Appendix A.

The immediate motivations for making this move are threefold. The first motivation is reliability. The normal lifetime of computer hardware is five years: The current ISI workstations are now seven years old. The support personnel are experiencing difficulty in keeping these machines operational.

The second motivation is the promise of increased hardware performance. My thesis advisor has estimated that the overall performance increase will be about 25%. The increased performance is important to both MDBS operations and the associated activities of researcher. For some applications, such as re-compiling large sections of the code, the new hardware will offer a welcome increase in productive time.

The third motivation is provided by the opportunity to utilize the better software available to users of SunOS on the Sparc architecture. The new platform will more easily support a much wider range of software useful to the researchers. This includes immediate X-Windows support plus access to better compilers and other tools.

D. THE ORGANIZATION OF THIS THESIS

This thesis is organized into seven chapters. Chapter I consists of this introduction. In Chapter II, we outline the approach taken to accomplish the porting and suggests ways in which this could have been improved. In Chapter III, we consider software issues important to the migration. In Chapter IV, we discuss the specific hardware issues in porting MDBS and describe the changes made in moving MDBS to the new platform. MDBS is communications-intensive, and in Chapter V we are concerned with these communications issues. In Chapter VI, a collection of other, smaller issues encountered in porting MDBS are presented. Finally, in Chapter VII, we conclude the thesis and outline problem areas and related issues requiring further study.

The chapters are supported by five appendices. These five appendices contain additional technical details, observations, and segments of code. In Appendix A, we provide a detailed comparison of the old and new platforms. In Appendix B, we provide information on the use of debugging flags, makefiles, and helpful scripts associated with MDBS. In Appendix C, a description of the structure and a listing of the files associated with the controller are given. In Appendix D, a similar listing for a standard backend is given. In Appendix E, the details about the sample database created to demonstrate the functioning of MDBS on the new platform are given. In Appendix F, the code for the new functions added to MDBS is provided.

II. THE APPROACH

A. AN INTRODUCTION

In a sizable project such as the porting of a large UNIX application, the importance of an organized approach to the porting task cannot be overemphasized. The software of Multibackend Database Supercomputer is both large and complex. MDBS consists of approximately 527 separate files distributed over 68 subdirectories with an aggregate size of about 23 megabytes (source code, object code, executables, scripts, and temporary files). The creation of the twelve processes (six in the controller, six in each backend) required to run MDBS is controlled by forty makefiles at different locations in the controller code. Each makefile contains information and instructions for compiling and linking the executable programs. On the top of the size and scope issues, there are issues of hardware delivery schedules and the interdependent time-tables of co-workers. This chapter presents the evolution of the porting approach utilized to port MDBS to Sun-4 workstations and concludes with some recommendations for future porting projects.

B. THE TASK

The successful completion of this project involves changing both hardware and software. The old ISI-V workstations, based on the Motorola 68020 processor and operated under the Berkeley UNIX (4.3 BSD), are to be replaced by Sun workstations, based on the Sparc architecture and operated under Sun UNIX (*i.e.*, SunOS 4.1.1). It is significant that the same Ethernet communications bus is to be used for the new system. A detailed description of the platforms involved may be found in Appendix A. The workstation hardware and operating-system software preparations are the responsibility of the support staff. Since MDBS requires only off-the-shelf hardware and system software, their work is not detailed here. This thesis instead concentrates on what has been done to move and adapt the MDBS software to the new platform. Simply stated, this involves relocating the code, modifying it to function as originally intended, and recompiling all the executables. To properly modify the student-and-researcher-written software is the heart of the problem.

C. GUIDING CONSIDERATIONS

Three considerations shaped the work on this project. First, the time is of the essence. Only a limited amount of time is available for the completion of the porting. Other researchers desire to use the enhanced capabilities of the new system. This consideration has limited modifications to those required to get MDBS up and operating on the new platform. Non-essential things which require improvement are documented, but such improvements must wait until later. A list of recommendations for further work may be found in Chapter VII. A listing of the two completely new functions added to MDBS during this porting project may be found in Appendix F.

The second consideration influencing the approach to this project has been the uncertainty as to the delivery date of the new hardware. The transfer of the desired hardware is contingent upon other acquisitions, and the likelihood of a delay has been real. This consideration has resulted in a conscious effort to emphasize the advanced preparatory work which could be accomplished prior to the delivery of the new hardware. The preparatory work consisted of a thorough analysis of the structure and functioning of MDBS, a study of the similarities and differences between the old and new operating systems, and familiarization and experimentation with the new system's compiler. This advanced preparatory work has paid off handsomely, resulting later in a relatively smooth implementation with a minimum of unforeseen difficulties.

The final consideration influencing the approach to this project has been the requirement to minimize the disruption to other researchers working on MDBS. Three other projects have been conducting research concurrently with the porting project. An approach is needed which could utilize the existing communication network during the preparation of the new system without stopping the work of the other teams on the old system.

These three considerations have influenced the porting plan which is described in the next section.

D. THE INITIAL PLAN

The initial plan was firmed up on February 23, 1993. It calls for a four-month preparatory period, followed by three months for the modification and one month for the testing and evaluation. The preparatory period involves the comparative study of both operating systems' implementation of data types, system calls, and communications. Differences between the old and new compilers, especially with regard to function libraries, are examined. This time also allows the selection of a specific version of the MDBS software to be ported (*i.e.* the "greg" version has been selected as the most advanced multibackend version). It is necessary to identify implementation dependent information, such as host names or data directory names, which are hard-coded into the software. The preparatory period is also used to prepare "clean" paper models of a backend and a controller free of the extraneous files left over from prior versions of the software. The preparatory time also allows an analysis of the new hardware (especially, the new, fixed disk drives) to assess any impact on the database system software. It is hoped that most every required change could be identified at this early stage.

The porting is to be accomplished in three phases to minimize the disruption to other researchers. Phase one, the preparatory work, is supported by the original MDBS configuration. Note that backends db5, db6, and db9 are crossed out or omitted in Figures

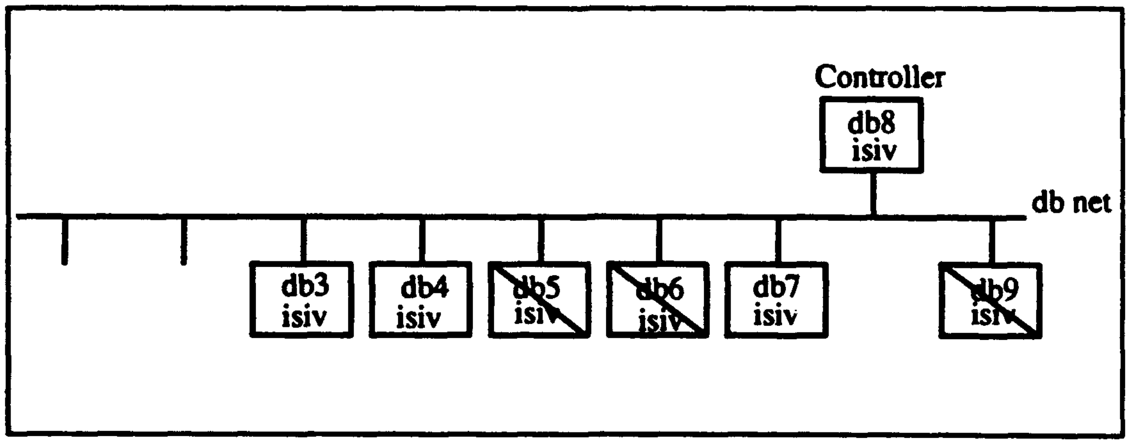

Figure 3. The Original MDBS Configuration

3 and 4, since they were not functional. The intermediate configuration of phase two would be achieved by simply adding the three new workstations (one new controller, two new backends) to existing connections on the network. MDBS will support any number of backends on the network. It also allows multiple controllers to be physically attached to the

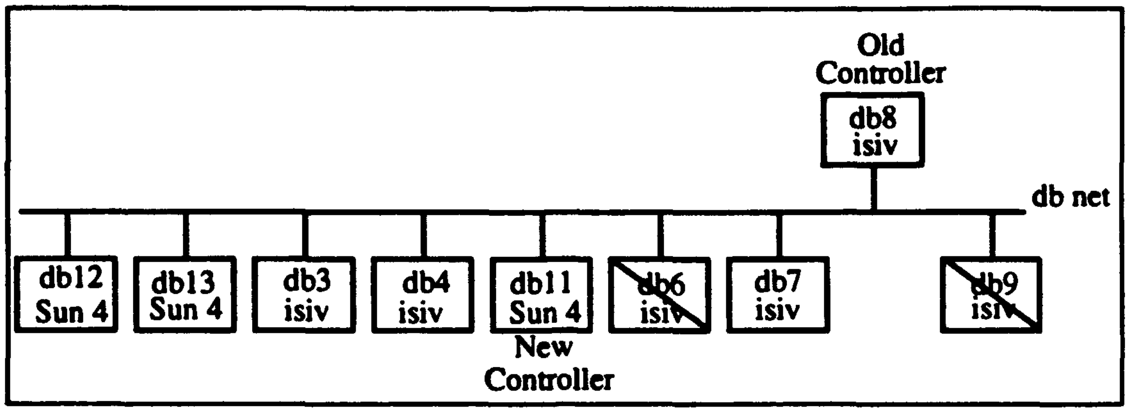

Figure 4. The Intermediate MDBS Configuration

network as long as only one of them is operating any of the six controller processes at a time (this is a function of the inter-machine communications design and port assignments). Users of the old system may continue to operate normally. The movement of files using the network remote copy command (rcp) and porting related runs of the new system may be executed whenever the old system is not in use. This arrangement maximized productive use of both systems. Once the porting work is completed and the new system is ready for use, the old workstations may be removed from the network. Note that the position of the

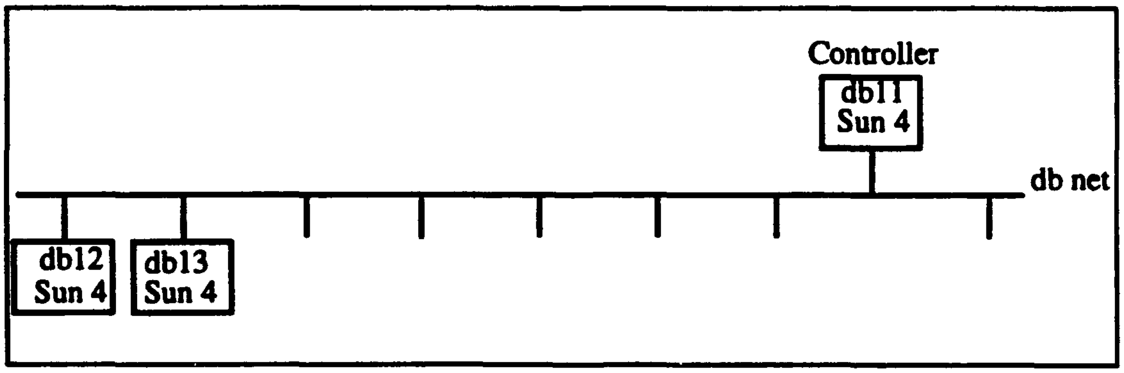

Figure 5. The Final MDBS Configuration

new controller is changed in Figure 5: This is because the primary contoller doubles as a communications gateway to another, unrelated network at this site.

The initial plan also focuses the initial work on the backend workstations, rather than the controller. This is due to the fact that the backends are much simpler and because the schedule called for the delivery of the backend hardware before the controller hardware.

E. MODIFICATIONS TO THE PLAN

The initial plan supports much of the ground work required before the actual porting can be started. The effort put into this analysis was time well spent. Like all first efforts, though, refinement was necessary once the actual work was commenced. A significant change involved the decision to develop the backends first. Backend-first development is possible, but not practical, because of the way MDBS compiles and distributes code. MDBS is designed so that all of the source code for both the controller and the backends are stored on the controller under the mdbs/VERSION/CNTRL and mdbs/VERSION/BE directories. A system of forty interrelated makefiles, also on the controller (see the file listing in Appendix C), handles the significant task of compiling the twelve processes required to run MDBS. Those six processes pertaining to each backend may then be manually copied to the mdbs/be.VERSION directory of each backend workstation or distributed automatically using the user interface documented in [Meeks, 1993, p. 26 - 27]. The small number of shell scripts and related files required for a backend can be copied from the old backend or from tape. See Appendix D for a listing of all files associated with a backend. Given the existing arrangement, it is more practical to begin the porting work on the controller and use its built-in capability to produce the backend executable files.

Another change to the initial plan was the length of time allocated to troubleshooting. The initial estimates were overoptimistic by a factor of two. This situation arose primarily because of the length of time required to address problems resulting from undocumented differences between the compilers, differing implementations of the shell programming languages, and the degree to which implementation dependent programming techniques were used in MDBS. These difficulties are documented in the chapters which follow. Additional facts and recommendations thought to be useful to the system porter are contained in Appendix B.

III. SOFTWARE ISSUES

A. AN INTRODUCTION

The software aspects of portability are defined in terms of the programming constructs employed and the capabilities of the target operating system and its associated compiler and library. Early incarnations of the Multibackend Database Supercomputer software existed on machines running the VMS and RSX operating systems. More modern versions have been built on 4.2BSD UNIX and, most recently, 4.3BSD UNIX. It should be possible to port MDBS to many operating systems supporting process control, reliable inter-process-communication, broadcast communication, and a suitable compiler. For the purposes of this porting project, MDBS will be moved to hardware running the SunOS UNIX operating system version 4.1.1. These operating systems are very similar in that SunOS is a derivative of BSD UNIX [Que, 1990, p. 18].

Developed prior to the creation of the ANSI standard for the C programming language, MDBS software is written in Kernighan and Ritchie standard C. The compiler used to create the executable files is the standard compiler (cc) included with the operating system. The use of implementation dependent programming constructs greatly increases the difficulty of any porting project. Fortunately, portability has long been a design consideration for MDBS [Wong, 1986, p. 9], and examples of non-portable code are infrequent.

In this chapter, we will discuss important issues and relevant differences between the original and target operating systems and between the old and new compilers. Issues related to communications will be discussed in Chapter V.

B. OPERATING SYSTEM CONSIDERATIONS

1. System Call Interface to the Kernel

In the UNIX environment, an application, such as MDBS, interacts with the hardware through a set of approximately one hundred system calls [Rosen and Rosinski and

Farber, 1990, p. 10]. These system calls instruct the kernel to perform various tasks, such as file I/O or process execution. The use of system calls by MDBS was the first operating system issue investigated as a part of this porting project.

The fewer and the more basic the system calls used by a UNIX application, the greater its portability [Rochkind, 1985, p. 16]. Only eighteen system calls are used to construct MDBS's higher level functions. These are listed alphabetically below, along with their location in the code.

Table 1: SYSTEM CALLS MADE BY MDBS

System Call	Purpose [ISIV, 1986]	Location
accept	accept a connection on a socket	pcl.c, sndrcv.c
bind	bind a name to a socket	ack.c, pcl.c, sndrcv.c
close	delete a descriptor (file or socket)	many places
connect	initiate a socket connection	pcl.c, sndrcv.c
exit	terminate a process	many places
gethostname	get the name of current host	bget.c, bput.c, cget.c, cput.c dbl.c
getnetbyname	get access to the network	pcl.c
getpid	get a process identification number	generals.c
gettimeofday	get the date and time	generals.c
kill	send signal to a process	shell scripts
listen	listen for connection on a socket	pcl.c, sndrcv.c
lseek	move the read/write pointer	cpcount.c, dio.c, dicp.c, rectag.c, zero.c
open	open a file for reading or writing	many places
read	read input (files or sockets)	cpcount.c, dio.c, disp.c, iig.c, meta.c, pcl.c, rectag.c, sndrcv.c
send	send a message from a socket	ack.c, cb.c, sndrcv.c, others

Table 1: SYSTEM CALLS MADE BY MDBS

System Call	Purpose [ISIV, 1986]	Location
socket	create an endpoint for communication	ack.c, pcl.c, sndrcv.c
unlink	remove directory entry (file or socket)	sndrcv.c, gsmodset.c
write	write output (file or socket)	bes.c, cpcount.c, dio.c, iigdbl.c, meta.c, pcl.c, rectag.c, sndrcv.c

We experienced only one problem as a result of differences between the implementation of system calls on the old and new operating system. The singular difference was the inability of the lseek() call to return the size of a raw device opened as a character special file. This technique was employed to return the disk size in the original version of zero.c. The information is passed to the user as an advisory. It is not critical to the zeroing function and was commented out of the new version.

2. Differences in the Shell

In UNIX, the shell is a command interpreter program (and programming language) that serves as an interface between the user and the operating system. The shell receives commands and arranges to have them executed. The shell scripts, or interpreter files (start.cntrl, run.be, stop.db*, zero.db*, etc.), supporting MDBS are designed to run on the C shell, a very common replacement for the original (Bourne) UNIX shell. MDBS needs a shell that supports job control since it must be able to specify that certain processes are run in the background. The C shell universally provides this support [Stevens, 1992, p. 248 - 249]. The SunOS provides the C shell, but some differences in its implementation were noted. The scripts governing the start-up of controller processes (start.cntrl), the start-up of backend processes (run.be), and the halting of running MDBS processes (stop.cmd) required modification to avoid syntax errors on the new system. The specific modifications are detailed in Appendix B.

3. Differences in Stack Implementation

Good programming practices are followed with the goal of writing implementation independent code. Where good practices are followed, minor differences in the underlying implementation are usually unimportant. Sometimes, the form of an offending statement is subtle. One such example was discovered during this porting project.

The add_path() function is a simple procedure located in the utilities.c file. It is called from twenty-seven places in the controller code. Its purpose is to append the path of the data directory (DATA_AREA) to the front of any file name passed to it so that a pointer to the entire construct can be passed to an open() system call. The form of this function used on the ISIV/4.3BSD platform had worked as intended for years. The function is reproduced below.

```
char *add_path(filename)
{
     char path [MaxPathLength + MFNLength + 1];

     strcpy(path, DATA_AREA);
     strcat(path, filename);
     return (path);
} /* end add_path */
```

This same function compiled but caused numerous run-time errors on the new Sun/SunOS system. The problem results because the storage for "char path" is created within the add_path function rather than in the original calling routine. The pointer to path which is returned to the caller is corrupted because of differences in stack implementation used by Sun. The problem was corrected by modifying add_path() to accept both the filename and the storage space for the path from the calling routine.

This is included here as an example of the subtlety of programming problems which arise because of operating system implementation differences.

4. Path Name and File Name Considerations

Persons involved in porting MDBS to other platforms should be aware of the following limitations coded into the current implementation. Commdata.def, located in the COMMON directory, limits the maximum length of file names (40) and path names (40). The configure.h header file, located in the "version (*i.e.* greg)/bin" directory allows backend and controller path names of length sixty-four. No changes were necessary for this porting project, but some operating systems may impose more restrictive limits.

C. C LANGUAGE COMPILER & LINKER CONSIDERATIONS

1. The C Language Library

Another source of concern for this porting project is compatibility of C language library header files between the old and new compilers. It is the header files which determine the availability of library functions, the names of symbols, the format of data structures, and the specification of communication sockets. Only those functions and definitions contained in header files referenced by the MDBS code are of concern, but significant differences in these areas could result in a greatly increased porting effort. The MDBS code references seventeen system-supplied header files. These are listed below alphabetically.

Table 2: HEADER FILES REFERENCED BY MDBS

included header files
arpa/inet.h
ctype.h
curses.h
errno.h
fcntl.h
math.h

Table 2: HEADER FILES REFERENCED BY MDBS

included header files
ndbm.h
netdb.h
netinet/in.h
stdio.h
strings.h
sys/file.h
sys/socket.h
sys/time.h
sys/types.h
sys/un.h
time.h

An analysis of these header files from the old and new systems revealed only minor differences. Examples which resulted in trivial changes to the code include the substitution of unsigned character for simple character types in stdio.h. Some of the Sun-supplied headers, including sockets.h, are actually licensed copies of the BSD code. Both sets of headers contain some "enhancements", but for the most part, these changes are not important to MDBS.

2. Type Conversion

C is not a strongly-typed language, but its type-checking capabilities have evolved over time [Kernighan and Ritchie, 1988, p. 3]. If the two compilers do not handle type-checking and type conversion (especially implicit type conversions) in the same manner, then compatibility problems could result.

Two classes of compile-time errors and warnings resulted from this difference in type handling. The first grew out of the use of the statement, FILE *open(), which was repeated at five locations in the original code (iig.c, iigdbl.c, bes.c, rectag.c, and dio.c). The original compiler allowed this redeclaration of the return type from the open() system call. The newer compiler did not. The impact was type mismatches in the way file pointers were obtained and used. The fix involved rewriting these routines to use file descriptors with open(), read(), write(), and close() calls rather than file pointers. This was preferable to using the fopen() library call to obtain file pointers which could be used with fread(), fwrite(), and fclose() library calls because of the increased speed of the unbuffered system calls.

A second class of compile-time problems resulted from differences in the handling of the automatic conversion of incompatible data types. The older compiler allowed, and the original MDBS code included, numerous examples of implicit conversion. One common case involved the interchange of an integer and a pointer. This practice was allowed in the original definition of C, but is no longer permitted [Kernighan and Ritchie, 1988, p. 3]. The second common case involved incompatible structure pointers. An example from beno.c had a pointer to one type of user-defined structure (type ctt_definition) pointing to a very different user-defined structure (type ciat_definition). The solution to both problems was the addition of careful explicit type conversion.

One curious example of implicit type conversion which did not raise an exception at compile-time but caused problems at run-time was discovered. The following statement was included in the source code for the backend put (BPUT) process:

```
printf("The host is %s\n", msg[2 * NoBElength + 2];
```

Since msg[n] (where n is an integer0 is a character, not a string, the %s should be a %c. This code compiled and ran acceptably on the old system. On the new system, it compiled without warning but caused the process to terminate at run-time.

3. Error-Checking

A newer compiler should have enhanced error-checking capabilities. This held true in the case of this porting project. The newer compiler identified three errors which had not been detected by the older compiler. Two of these involved missing macro names. MDBS source code includes a number of debugging statements whose inclusion is controlled by #ifdef statements. Two of these were undefined, meaning that those sections of debugging code had never been included. The third error involved an unmatched parenthesis.

4. The Linker and the Makefiles

The linker performs complicated actions on the object modules under the direction of the user-supplied makefiles. A wide range of compatibility problems are possible. First, the newer Sun operating system supports dynamic linking (run-time linking), something not supported by the older BSD compiler. The impact of this on the MDBS code was unknown. This feature seems to have no impact on the functioning of the MDBS software.

Certainly, a careful review of the compiler and linker options (flags) specified in the makefiles is in order any time an application is moved from one platform to another. Many of these flags are nearly universal in meaning, but others are implementation specific. The original MDBS makefiles specified the -20 option. This instructs the compiler to optimize code for the Motorola 68020 processor. This is inappropriate for the new system and would cause compile-time errors. These were removed from all forty makefiles.

One unexpected error which did surface involved the linking of the record processing (RECP) executable. Possibly because of its large size, the makefile directed the linking of recp.exe in two steps on the older system. This produced a program which always generated a segmentation fault and dropped core when executed on the new system. The problem was corrected by modifying the makefile to allow one-step linking.

D. IN CONCLUSION

In this chapter, we have presented seven classes of operating system and language/ compiler issues relevant to the porting of the Multibackend Database Supercomputer software. In the next chapter, we will consider issues relevant to hardware differences of the new platform.

IV. HARDWARE ISSUES

A. AN INTRODUCTION

Unlike some other database machines [De Witt, 1979, p. 122 - 132], the Multibackend Database Supercomputer was designed from the start to utilize only off-the-shelf hardware and identical standard system software. The positive result is a mercifully small number of hardware considerations relevant to the task of porting the software to a new hardware platform. Two significant hardware issues remain, those related to the optimal use of the fixed disks, and those related to the storage of hardware specific information within the code.

B. FIXED DISK ISSUES FOR MDBS

By definition, database machines are not general-purpose computers. Database machines are specialized for the tasks they must perform. A principal distinguishing characteristic is the capability to handle very large amounts of stored data quickly. The MDBS architecture emphasizes the efficient data handling capabilities of large, fast disk drives in each backend machine.

1. Fixed Disk Requirements

A typical MDBS configuration would involve one fixed disk drive in the controller and three fixed disk drives in each backend machine. The controller drive is mounted (attaching the file system to the directory structure) so that it can be used to store and execute the MDBS programs in the normal manner. Any fixed disk with at least twenty-three megabytes of usable capacity will suffice. One of the three backend drives should also be mounted. This disk need only store the approximately two megabytes of code required by a backend machine. The other two backend drives should not be mounted, as these will be accessed as raw devices. The size of the first of these two raw drives, the base data disk, will be determined by the amount of data to be stored divided by the number of backends over which it can be shared. The second backend disk, the meta data disk, should be about one-fourth the size of the base data disk. In the Naval Postgraduate School's Laboratory for Database Systems Research the base data drives have been as large as four hundred megabytes to as small as one hundred megabytes each. Irregardless of the capacity, it is important that all of the attached backends have identical base data drives and identical meta data drives. The following paragraphs will make the reason for this requirement clear.

2. Understanding MDBS Disk Utilization

Understanding how MDBS utilizes it's fixed disks is fundamental to the rest of this presentation. As stated above, the controller machine's disk and each backend machines' program disk are mounted drives. These are used to store the source code, executable code, shell programs, transaction files, and temporary files listed in Appendices C and D.

The meta disk and data disk are unmounted drives. They remain raw devices so that they can be used as character special files by the MDBS software. In essence, the entire fixed drive is viewed as a special kind of file by the operating system. It is not uncommon to use this approach for database applications on UNIX systems, since it allows the database management system (DBMS) to by-pass the file system and directly transfer data between the processes' address space and the fixed disk using direct memory access [Rochkind, 1985, p. 3 - 4]. The result of bypassing the buffered file system is greatly improved performance.

The significant performance gain possible with character special files does not come without a price. Raw devices cannot use most of the convenient file handling utilities provided by the operating system. It is necessary to write the routines to handle the storage and retrieval of data on the disk. Frequently, these routines are gathered together into a collection of subroutines which can be called directly by the operating system kernel. This is referred to as a device driver [Rochkind, 1985, p. 4]. The routines for accessing the meta- and base-data disks on MDBS are not unified into a device driver. Rather, they are contained in the source code for the directory management (DM, or DIRMAN) process, which handles meta-data disk access, and the disk input-output (DIO) process which is responsible for access to the base-data disk.

3. Implementation Details of the Meta Disk

Meta-data describes the structure of the primary information stored in the database. The meta-data disk stores this information about the base (primary) data using a special format designed to take advantage of the speed of raw devices used as character special files.

The beginning of the device contains two global tables, known collectively as the "header". The first of these tables, the "Next Available Track Table" (NATT) is used to store the information about the next available (unused) disk track on the data disk which may be used for starting a new data cluster. The NATT starts at byte zero of the device. The other table, the "Offset Table" (OT), is a list of database identifiers for databases which have data stored on any backend machine. The OT begins at byte three of the device. The entire header is of fixed size. The current value is 500 bytes. All of the values quoted here are stored in the meta.def file located in the DM directory.

The header area is followed by space for one or more sets of tables. Each set of six tables contains information about a single database. These six tables are the Descriptor-to-Cluster Bit Map Table (DCBMT), the Templates list, the Attribute Table (AT), the Descriptor-Descriptor ID Table (DDIT), the Descriptor Table (DT), and the Cluster-Definition Table (CDT). Each table has a pre-determined length specified in the meta.def file.

```
#define NATT_OFFSET  0
#define OT_OFFSET    3
#define HEADER       500
```

Figure 6, on the following page, shows the organization of the meta disk. Identical meta-data tables are maintained on each backend machine. Each meta-data disk contains a complete set which is identical to every other.

Item	Contents	Starting Address
Header Block	NATT	NATT_OFFSET
	OT	OT_OFFSET
Meta-Data Block for 1st Database	DCBMT	HEADER+(DB_no*DB_LENGTH)+DCBMT_OFFSET
	Templates	HEADER+(DB_no*DB_LENGTH)+TEMP_OFFSET
	AT	HEADER+(DB_no*DB_LENGTH)+AT_OFFSET
	DDIT	HEADER+(DB_no*DB_LENGTH)+DDIT_OFFSET
	DT	HEADER+(DB_no*DB_LENGTH)+DT_OFFSET
	CDT	HEADER+(DB_no*DB_LENGTH)+CDT_OFFSET
The meta-data blocks for other databases follow sequentially		

Figure 6. Meta-Data Disk Storage Format

Once the disk is opened using the open() UNIX system call, information pertaining to a database may be written (write()) or read (read()) from the disk. The location of the meta information may be simply calculated by adding the header size to the offset needed to reach the desired table in the *nth* database, where n is the numerical representation of the order of the database ids contained in the Offset Table. The routines for opening and using the meta-disk are contained in the meta.c source file located in the DM directory. The directory management process, running on each backend machine, carries out these activities at run-time.

4. Implementation Details of the Base-Data Disk

The base-data (primary data) is the actual data of interest to the user of the database management system. The base-data disk is also configured as a character special file, but here the similarity ends. The base-data is stored on the base-data disk in a manner completely different than that of the meta-data. The base-data storage scheme is designed to spread the records of a database across the backends as evenly as possible. Base-data information is not replicated. On each backend machine's base-data disk, records are stored in one or more tracks of the same disk cylinder. During a retrieve operation, all of the attached backend machines would be performing their reads at the same time. This arrangement achieves a primary high-performance design goal for MDBS, *cylinder parallel readout* of data [Hsiao, 1991, p. 50 - 53]. Figure 7 depicts how four records (x), residing on different surfaces of the same cylinder, may be efficiently read from a data-disk drive.

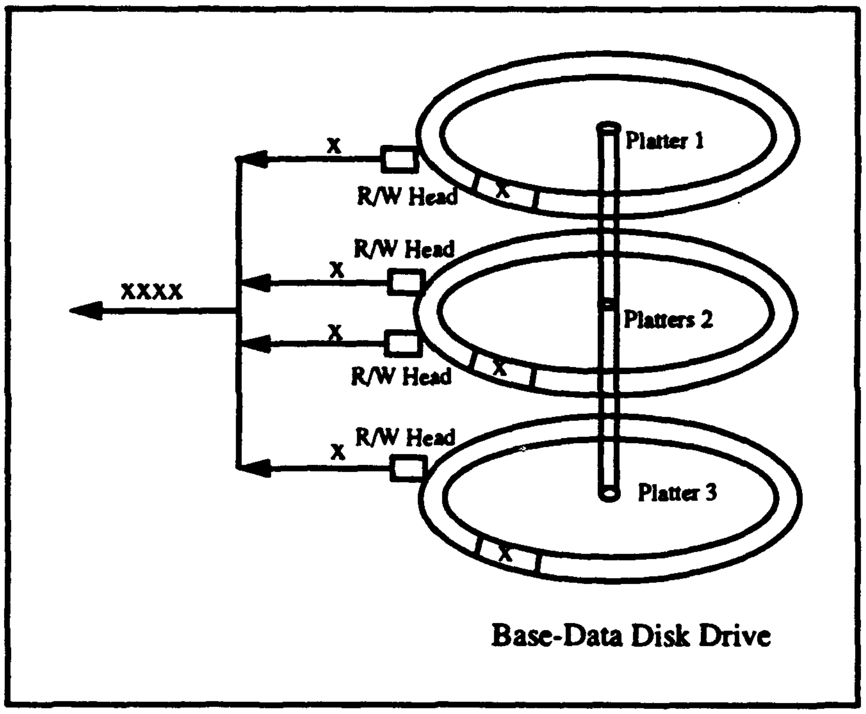

Figure 7. Record Distribution on a Base-Data Disk Drive

Unlike the meta-data storage scheme, which was based on the simple calculation of byte offsets into the device, the complicated base-data storage scheme relies on an understanding of several physical disk parameters (track size, tracks per cylinder, number of cylinders) to arrange the records into the desired tracks of the disk for optimal reading and writing. Again, the basic storage unit of the base-data disk is the track, not the byte.

This dependence on knowledge about physical disk details makes the code for the base-data disk the most implementation dependent of all of the MDBS code. Fortunately, this information has been gathered into three header files which are listed in sub-section 7 below.

The routines for opening, reading, and writing the base-data disk are contained in the disk input-output (DIO) directory and executed at run-time by the DIO process. Each backend machine knows its next available cylinder and track (if any). Decisions as to which backend machine will receive a newly inserted record are made by the insert-information generation (IIG) process on the controller after coordination with the backend DM processes [Boyne and Demurjian and Hsiao and Kerr and Orooji, 1983, p. 29].

5. Disk Initialization

Before their first use, both the meta-disk and data-disk must be initialized. The initialization process is repeated whenever a database is removed from the system. This process involves opening the devices and writing null zeros to all or a portion of the device. For MDBS, this process is referred to as "zeroing" the disk (Note that only meta- and data-disks should be zeroed - never zero one of the program disks, as this destroys all of the code stored on the disk!).

Initialization is handled by an MDBS utility named "zero". The executable code for zero should be copied to the "mdbs/bin" directory on each backend. A master copy of the zero executable is maintained in the "version (*e.g.*, greg)\bin" directory on the controller. The source code for zero (zero.c) is located in the "DIO" directory on the controller. Zero is normally called and passed necessary information (device to zero, number of bytes to zero) by the shell programs located in the "run" directory of the controller.

6. The Problem and the Solution

The aforementioned methods for initializing, writing, and reading data work well on the old hardware platform. The old fixed disk drives utilize the *Enhanced Small Device Interface* (ESDI). The disks' formatting information is stored on the drive controller, and the entire surface of the disk is available for use by MDBS.

The new hardware platform is equipped with disk drives utilizing the *Small Computer Systems Interface* (SCSI). SCSI is a system level interface, meaning that some of the controller functions must be built into the drive circuitry [Rosch, 1989, p. 550]. When one of the new SCSI drives are operated under the SunOS 4.1.1 operating system, the formatting information for that drive must be written to the disk. Sun refers to this information as the "label". It is located in the first block (cylinder 0, head 0, track 0) of the disk. This represents an implementation dependency with respect to the existing MDBS code, since the existing routines for initializing and writing to both the meta-data and base-data disks destroy the formatting information stored in the label block on the new drives.

With regard to the disk initialization routine contained in zero.c, the solution is straight-forward. The old version of zero starts at the first byte (location 0) of the drive and writes null zeros. We have added a new constant, called "SAFETY_OFFSET", which is currently defined in meta.def to be 512. The lseek() system command which zero.c uses to position the file pointer, now begins writing null zeros at location SAFETY_OFFSET, rather than at location 0. The label block of the drive is passed over and preserved.

The modification to the meta-data disk writing routines is also straight-forward. The old routines write the header information beginning at location 0 on the meta-data disk. To avoid destroying the label information, the same SAFETY_OFFSET is added to the meta-disk before the header block. This is accomplished by adding a new definition (SAFETY_OFFSET) to the meta.def header file and "sliding" the other blocks further into the device. The advantage of this approach is that it is transparent to the rest of the existing code. The routines for the calculation of table locations reference the definition of HEADER, so only the following changes are necessary:

```
#define SAFETY_OFFSET         512
#define NATT_OFFSET           0 + SAFETY_OFFSET
#define OT_OFFSET             3 + SAFETY_OFFSET
#define HEADER                500 + SAFETY_OFFSET
```

Figure 8 shows the new organization of the meta-data disk.

Item	Contents	Starting Address
Safety Offset	Label	0
Header	NATT	NATT_OFFSET + SAFETY_OFFSET
	OT	OT_OFFSET + SAFETY_OFFSET
Meta-Data Block for 1st Database	DCBMT	HEADER+(DB_no*DB_LENGTH)+DCBMT_OFFSET
	Templates	HEADER+(DB_no*DB_LENGTH)+TEMP_OFFSET
	AT	HEADER+(DB_no*DB_LENGTH)+AT_OFFSET
	DDIT	HEADER+(DB_no*DB_LENGTH)+DDIT_OFFSET
	DT	HEADER+(DB_no*DB_LENGTH)+DT_OFFSET
	CDT	HEADER+(DB_no*DB_LENGTH)+CDT_OFFSET
The meta-data blocks for other databases follow sequentially		

Figure 8. Modified Meta-Data Disk Storage Format

The changes required to protect the label area on the base-data disk require a different approach. It does not make sense to try to follow a byte-based approach to protecting the label block, since the existing storage routines are based on cylinders and tracks. Likewise, an approach which tests every insert operation for indications that a write to the label block is about to occur is not satisfactory since this would impose the overhead of two extra comparisons per write.

The Next-Available-Track-Table (NATT) on the meta-data disk stores the next available cylinder and next available track information for the backend on which it resides. Whenever a backend does not already contain records from an existing database, these values are both zero. This is because the initialization routine (zero) writes zeros to every location on the disk. Two global external variables, av_cylinder (type unsigned short) and av_track (type unsigned char) are declared in dirman.def for the purpose of receiving and holding these values during execution. The information is read into the variables during an initialization call to the read_meta_NATT() function in the DM process.

Since our goal is to preserve the label area of the disk, the obvious way to proceed is to ensure that the available cylinder and track variables never indicate that cylinder 0 track 0 (the location of the label) is available for writing. Rather than modify the routine which zeroes the base-data disk, we will add a new function, init_meta_NATT(), to meta.c which will write a zero to the available cylinder and a one to the available track portion of the Next-Available-Track-Table immediately after the meta-data disk is opened. This call will execute before the read_meta_NATT() call so that the global variables receive values of zero and one, rather than zero and zero.

The first write to the base-data disk now skips cylinder 0/track 0 (where the label area is located) and writes instead to cylinder 0/track 1. This approach costs one track per backend base-data disk, but avoids the complexity of a byte-based scheme and the run-time overhead of testing before writing. It also has the advantage of being flexible, in that the base-data disk's beginning cylinder/track address can be changed to suit the user's need simply by changing the values of the new constants "first_record_cylinder" and "first_record_track" contained in dirman.def.

7. Hard-Coded Disk Information

As discussed above, certain implementation-dependent fixed-disk information is hard-coded into MDBS. All of the fixed disk related definitions are contained in four header files. The following definitions are from the commdata.def file in the COMMON directory:

```
#define RecDiskSize    95421000
#define no_tracks      6
#define TrackSize      8192-2
```

RecDiskSize refers to the size of the new base-data (record) disk. The definition for no_tracks is the number of tracks per cylinder (the number of heads) on the new disk. TrackSize is the segment size for the storage of records on each track of the base-data disk. It is included here because this was originally the same as the formatted track size of the disk, but this changed as a result of performance testing several years ago. Changes to this value should be carefully considered, as there are numerous side effects.

The dio.h file in the DIO directory contains the device name for the base-data (record) disk in the following statement:

```
char *driver_names = { "/dev/sd4c"};
```

This same value is written into the code for the disk utility rectag.c, and each of the zero.db* shell scripts located in the "run" directory.

The following definitions are contained in the meta.def header file (these are in addition to the structure information stated earlier):

```
#define META_DISK_NAME      "/dev/sd2c"
#define META_DISK_STORAGE   95421000
```

The META_DISK_NAME is the device name of the meta-data disk. The value specified for META_DISK_STORAGE is the formatted storage capacity of the meta-data disk.

The final header file containing fixed disk specific information is dirman.def in the DM directory. It contains the following definition:

```
#define no_cylinders 974
```

This is the number of cylinders on the base-data (record) disk.

C. OTHER HARDWARE SPECIFIC ISSUES FOR MDBS

While the physical details of the fixed disk drives are the most obvious hardware-specific information coded into MDBS, there are other details important to a porting project. Details of the workstations and network hardware comprising MDBS must be considered when moving to a new hardware platform.

1. Workstation Information

One major area of concern involves the storage and use of information (*e.g.*, host names) relative to the MDBS workstations. The maximum number of workstations attached to the MDBS ommunication network, their specific names, and the format (alphabetic characters and numbers) of their names are all hard-coded into MDBS. The maximum number of backend machines is explicitly coded in two places (once as an integer, once as a character) in pcl.def (located in the COMMON directory). The character version of the number is not used by the "greg" version of the software. The maximum number of backend workstations is implicitly defined in another place in the code. This statement from ti.c (located in the TI directory) sets a condition for a successful start-up of the system:

```
if (NOBE[0] > '0' && NOBE[0] < '3') {
```

This number should be one greater than the maximum number of backend machines. The dbl_template() function in the dbl.c source file contains several statements which are highly dependent on the number (and name) of the workstations comprising MDBS. The values associated with the variable buMP and it's length (buMPno) are critical. This function should be carefully analyzed by anyone porting MDBS to a new hardware platform, as it is very implementation dependent.

A complete set of the host (workstation) names are written into the configure.h file. Configure.h is the header file for the executable (main) which automates the start-up and shut-down of the MDBS software. The host name of the controller machine is specified separately in configure.h and also contained in pcl.def.

Some routines in the MDBS code are sensitive to the length of the host names of the attached workstations. These are routines which pick the names of sending stations from messages by relying on the length of the station name.The very implementation-dependent coding of the dbl_template() function in dbl.c has already been mentioned. Another example is found in ack.c (from the COMMON directory), which includes this statement at about line 626:

```
for (i = 0; i < 4; i++)
   loudmouth[i] = hp -> h_name[i];
```

The number 4 represents the number of characters (alphabetic and numeric) in the host names (*e.g.*, db13 is four characters long). The number 4 has subsequently been replaced by the definition host_name_length. A similar example is written into pcl.c (also located in the COMMON directory). Here the implementation dependent statement in the get_first_message() function is:

```
for (i = 0; i < 4; i++)
      brdcstng_host [i] = hp -> h_name[i];
```

Here, also, the number 4 results from the number of characters in the host names. Because of statements like these, all of the host names of the machines making up MDBS should have the same length. The current implementation is set up for host names of length four, but this could be changed. The constant MAX_HOST_LEN, contained in msg.def (located in the COMMON directory), could be used to describe the length of host names.

Some sections of MDBS code are sensitive to the format of the host names (*e.g.*, "db11"). Ack.c, located in the COMMON directory, is a principal source file for routines which acknowledge messages broadcast by the controller or a backend machine to all other machines. Some details about the implementation of reliable broadcasting for MDBS are located in [Wong, 1986, p. 38 - 44, 61]. The routines of this file are very highly dependent on the composition of host names. As originally implemented, these routines expect to deal with host names of length three, where the last character is a unique number which can be converted into an integer. The original code in ack.c also assumed that the highest number portion of the host name would never be larger than the maximum number of workstations attached to MDBS. We have made those alternations necessary to make the code easier to port to different hardware with host names of other lengths. New constants have been added (*e.g.*, lowest and highest numbered workstations) to logically separate the range of the number portion of the hostnames from the range of index values for the host_names array (the host_names array is a two-dimensional array which holds the host names of all workstations participating in a given run of MDBS). This separation improves portability and allows the efficient use of higher-numbered workstations. An illustration of this is provided in the assemble() function located in the ack.c source file. These additions also improve readability.

The new definitions include host_name_len (the length of the host name string), min_ws_number (the number portion of the least workstation host name), max_ws_number (the number portion of the greatest workstation host name). The values associated with these definitions, located in ack.def and ack.dcl, would have to be adjusted for a different set of workstations with different host names. The current implementation is set up for a hostname of length four, of which the last two characters are numbers. No two hostnames should end in the same numbers as MDBS uses these to uniquely identify each workstation.

A new function, host_name_integer(), which receives the hostname string and returns the number portion as an integer has been added to ack.c as a replacement for the old method. The source code for the new, more portable function is contained in Appendix F.

Almost all of the shell scripts associated with starting and stopping MDBS contain host name or device dependent information. These simple files, located in the "run" directory on the controller, include all of the stop.be*, zero.db*, and stop.db* files. All will require slight modification when MDBS is moved to a new hardware platform.

2. Network Communication Information

The third major source of hard-coded, implementation specific details in the MDBS code is the communication network. The name by which the network may be accessed and important hardware port assignments are specifically written into the code. More will be said about this in the next chapter.

D. IN CONCLUSION

In this chapter we have brought forth the three primary sources for hardware-specific statements in the MDBS code. A specific modification to the way in which the raw devices are written has also been described. In the next chapter, we will present details of the communications issues important to porting MDBS to a new platform.

V. COMMUNICATIONS ISSUES

A. AN INTRODUCTION

The third major issue which defines the size and scope of the porting task for anyone moving the Multibackend Database Supercomputer to a new hardware and/or software platform is communications. The loosely coupled parallel architecture of MDBS is very highly dependent upon communications. This communication requirement exists between the processes running on any one workstation and between processes running on different workstations. This chapter starts with a presentation of the communication requirements of MDBS. Next, the design of MDBS communications is discussed. Finally, details relevant to specific implementations are provided.

B. MDBS COMMUNICATIONS REQUIREMENTS AND IMPLEMENTATION

MDBS requires both inter-process and inter-machine communications support. Inter-process communication (IPC) is communication between the processes running on a single workstation. The controller depends on reliable inter-process communications to coordinate the actions of the six processes running concurrently on the controller workstation. Each backend machine depends on reliable inter-process communication to coordinate the actions of the six backend processes.

1. Inter-Process Communications

The current implementation of MDBS supports inter-process communication through the facilities offered by the old (BSD4.3) and new (SunOS 4.1.1) UNIX operating systems. MDBS uses sockets of type SOCK_STREAM in the UNIX domain (AF_UNIX) under the Transaction Control Protocol (TCP) for communications between processes on the same machine. The TCP communications protocol is reliable, meaning that there is no need for MDBS to check for the delivery, sequencing, or duplication of messages sent using this protocol. A message which is transmitted may be assumed to be delivered successfully. This is of great importance to a database system where data integrity is a central concern.

An inter-process communications channel is established asymmetrically using the client-server model [Leffler and Fabry and Joy and Lapsley and Miller and Torek, 1987, p. PS1: 8-2 - 8-10]. The client makes the system call, socket(), which creates an endpoint for communication and returns a descriptor. The client then attempts to connect to a server using the connect() system call. The server also creates a socket and then uses the bind() system call to assign a name to the socket. The server then listens for a connection attempt by the client process. Once the connection has been established, simple read() and write() system calls may be used to transfer data. These messages are not limited to a specific length by the operating system. These messages are written into buffers created by MDBS. The size of the MDBS buffers, not the operating system, limits the maximum size of the messages.

2. Inter-Machine Communications

Individual MDBS workstations need to be able to send messages to other individual machines as well as broadcast a message to all other workstations (backend and controller alike). The current implementation of MDBS also uses the support offered by UNIX for inter-machine communications. MDBS workstations are all connected to a standard ethernet cable. MDBS creates sockets of type SOCK_DGRAM in the Internet domain (AF_INET) under the User Datagram Protocol (UDP). The sockets of type SOCK_DGRAM support connectionless, unreliable messages of a small, fixed length [Rieken and Wieman, 1992, p. 39 - 45, 51]. By unreliable, we mean that messages may become lost - it is not possible to assume that a message will be received. Since reliability is critical to this application, another level of communications protocol was added to MDBS to support reliable broadcasting with acknowledgments. Details of this are contained in [Wong, 1986, p. 38 - 44, 61].

The familiar socket() system call is used to create the socket in the Internet domain. The connect() system call is used to establish the link. The socket's address includes the Internet address and port number. Messages are transmitted to another socket of type SOCK_DGRAM using the send() system call. The 32-bit Internet address is automatically assigned to all messages destined for this socket. Since the messages are of fixed format and length, MDBS uses the read() system call rather than the recv() system call to read the messages.

C. COMMUNICATION CHANNEL DESIGN AND FUNCTION

Now that the general form of MDBS communications requirements have been presented, a discussion of the design and use of communications channels within MDBS is needed.

1. Process Functions

A brief review of the functions performed by the twelve MDBS processes is included here as an aid in understanding the communication channel design. More detailed information about these processes and how their functions have changed over time is available in [Boyne and Demurjian and Hsiao and Kerr and Orooji, 1982, p. 3, 29 - 33], [Wong, 1986, p. 10, 38 - 44], [Hammond, 1992, p. 4 - 5]. The six controller processes are controller get (CGET), controller put (CPUT), test interface (TI), request processing (REQP), insert-information generation (IIG), and post processing(PP). The CGET process is responsible for sending DGRAM messages across the ethernet to other MDBS workstations. The CGET process is responsible for receiving DGRAM messages from other workstations. The TI process is the user interface. It contains the routines for activating the selected language interface and capturing the user's instructions from the terminal. The REQP process parses the user's requests and checks for proper format and syntax before forwarding the request. The IIG process handles the clustering of the database records across the backend machines. It includes a global table of locality information (backend number, cylinder, track). The PP process properly formats the results received from backend machines so that they can be displayed to the user.

The six backend processes are backend get (BGET), backend put (BPUT), record processing (RECP), concurrency control (CC), directory management (DM), and disk input/output (DIO). Au six of these processes run on each backend machine participating in MDBS. The BPUT process is responsible for sending DGRAM messages to other MDBS workstations over the ethernet cable. The BGET process receives these same inter-machine messages for the backend machine on which it is running. The RECP process is responsible for the manipulation of records. This includes selection, retrieval, and value extraction. The CC process is charged with maintaining meta-data and base-data (record) integrity during the processing of transactions. The DM process is responsible for all access to the meta-data disk. It coordinates with the record processing process in gathering information about how the base-data (records) are stored. The DIO process handles all reading from and writing to the base-data (record) disk.

2. Design of the Communications Channels

To support the inter-process and inter-machine communications requirements of MDBS, the communications links shown in Figure 9 (on the following page) are established using the system calls covered in part B. Figure 9 shows all of the channels built into MDBS, not just the primary ones. Only one backend machine is shown. It is representative of all backend machines. Note that the inter-process communication links have arrows. This is intended to show which process initiates the link, not the direction of information flow, since the SOCK_STREAM connections are bi-directional. The arrows associated with the SOCK_DGRAM sockets in the PUT and GET processes do not imply an actual connection, but do show the direction of message flow.

3. How MDBS Establishes Communications

All of the communication channels shown in Figure 8 are established during system generation (start-up). The MDBS code logically separates the establishment of inter-process and inter-machine communication. The inter-process communications are established within each workstation first. The IPC links within each machine are handled by the send-receive initialization routine (initsr()) contained in the sndrcv.c source file located in the COMMON directory..

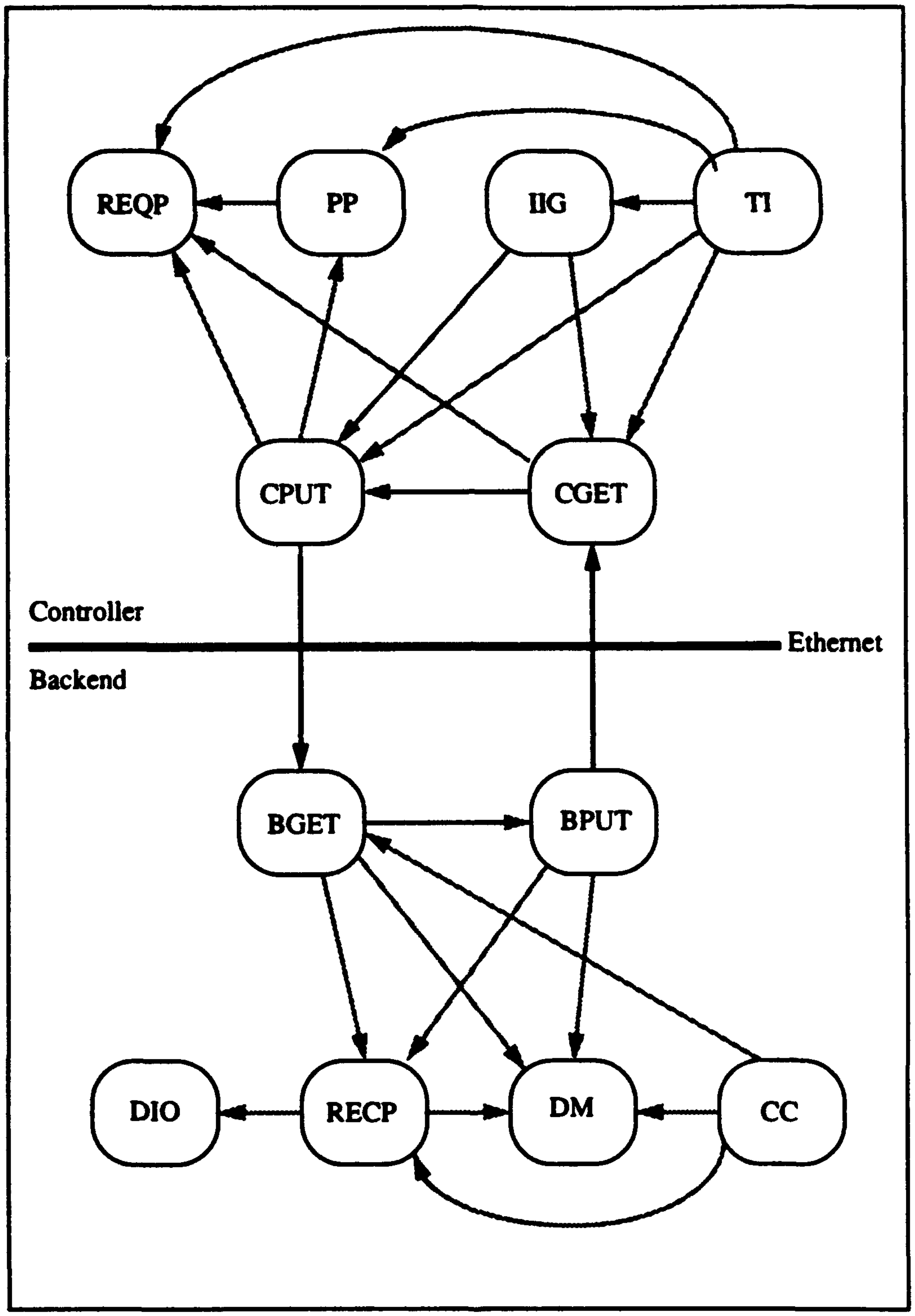

Figure 9. MDBS Communication Channels

Once the inter-process channels are established, the inter-machine links are created. The initcb() (initialize controller-backend communication) function in the cb.c source file and other functions in the pcl.c source file (both located in the COMMON directory) handle the establishment of inter-machine communication. Within initcb(), the initbtoc() function sets up the connectionless link and the init_ack_put() function sets up the additional reliability layer built into MDBS which allows reliable broadcasting. Each backend machine needs to identify itself to the controller before the controller's permanent socket is created. This is handled through the creation of a temporary, "universal" socket in the controller (using the init_serv() function in pcl.c). The universal socket is replaced by the permanent, dedicated controller socket once the controller knows how many backend machines are presently configured.

The job of establishing the inter-machine communication channels start in the controller. During initialization, the TI process tells the CGET process to set backend numbers with a SetNoBE (message code 923) message. The CGET process then receives an initial identification message (BeWho, message code 925) from a backend over the ethernet. CGET forwards this message to the CPUT process, which transmits a message (SetNoBE, message code 925) back to the identified backend over the ethernet. Initialization is complete when that backend's BGET process sends an inter-process message with the backend number (SetBeNo, message code 924) to the CC, DM, RECP, and BPUT processes. These and other process and message related codes are contained in the msg.def header file.

Shutdown is accomplished by the finish send-receive (finishsr()) function and the close socket (closesock()) function. Calls are made to these functions when system shutdown is ordered by the user or a critical communications failure occurs. Note that the inter-process communication sockets are unlinked immediately following creation. This does not close the socket as long as the associated process remains viable. This does ensure the socket is not left open if the process terminates abnormally [Stevens, 1992, p. 96].

D. DETAILS IMPORTANT TO PORTERS OF MDBS

This section contains communications-related information of particular importance to anyone porting MDBS to another platform.

1. Limitations on Message Lengths

There are limitations to the lengths of both SOCK_STREAM and SOCK_DGRAM messages of which the porter needs to be aware. The maximum length of the inter-machine messages handled by SOCK_DGRAM sockets is set by the operating system. UNIX limits such messages to 1450 bytes. This limit is hard-coded into the MDBS code as the constant BRDCSTSZ (broadcast size) in pcl.def (located in the COMMON directory). This value might have to be changed on a different operating system platform.

The maximum length of the inter-process messages handled by SOCK_STREAM sockets is not determined by the operating system, but rather by the amount of buffer space set aside by MDBS for messages. The constant, MSGLEN (contained in both msg.def and licommdata.def), limits inter-process messages to approximately eight kilobytes. This value is related to the track size of the base-data disk. This limit might have to be changed on a different platform. If a change to the buffer size is necessary, the values of the following definitions will also have to be changed: RESTMSGLEN (located in msg.defin the COMMON directory), PP_ResBufSize (located in pp.def in the PP directory), RESLength and REQLength (both located in tstint.def in the TI directory).

2. Access to the Network

MDBS must be able to gain access to the network and communication ports in order to implement inter-process and inter-machine communication. Access for inter-process communication is considered first. The pcl.def file, located in the COMMON directory, contains the hard-coded name of the network joining the MDBS workstations. The constant NETNAME (currently "npsisnet") is the name by which MDBS accesses the network. The getnetbyname() system call, issued from pcl.c, establishes the access.

Whatever network name is used, it must be reflected in both pcl.def and the UNIX operating system's network database. This database is stored in the /etc/networks file.

Inter-machine communications. in the Internet domain, requires additional, implementation-dependent information. Setting up the inter-machine communications requires access to the system's network database file, just as the inter-process communications did. This is because the network's 32-bit internet address is recorded there. Communications in the Internet domain also require a port number. MDBS code includes the hard-coded numbers for four ports. Pcl.def defines the port numbers for the backend and controller ports:

```
#define BE_PORT       1650
#define CNTRL_PORT    1651
```

Ack.dcl defines the port numbers for the retransmission and acknowledgment ports:

```
#define RETPORT       1700
#define ACKPORT       1800
```

These port numbers must be assigned by the system administrator and will be different in every implementation.

3. Socket Definitions

The operating system's definitions for the sockets it supports are contained in the sockets.h header file. The old and new operating system's versions of this file should be carefully compared. For this project, no changes were needed.

4. The Socket Directory

The location for the controller and backend sockets is specified in the MDBS code. The sndrcv.def header file includes the following definition:

```
#define PREFIX        "/u/mdbs/Sockets"
```

Any change to the location for the sockets in the controller or backend directory structure must be reflected here.

5. A New Aid for Timing Messages

An additional debugging flag has been added to the existing ones in the flag.def files to allow easier reading of the process trace files. This flag is the send-receive timing flag (SRTimeFlag). When included in the flag.def files, it causes the system time (in seconds) to be added to the trace whenever any inter-process communication functions (send() and receive()) or inter-machine communication functions (put_message() and get_message()) are called. This timing information can be useful for understanding or trouble-shooting the traces belonging to the six processes running on a single workstation, but are of limited use in making comparisons between different workstations since workstation clocks are rarely so closely synchronized.

E. IN CONCLUSION

Communications, both inter-process and inter-machine, are central to the functioning of the Multibackend Database Supercomputer. Any attempt to move MDBS to a new hardware or operating system software platform must carefully consider the communications support available on the new platform as well as communications parameters hard-coded into MDBS.

VI. MISCELLANEOUS OTHER ISSUES

A. AN INTRODUCTION

The purpose of this short chapter is to bring out two issues which are important to porting the Multibackend Database Supercomputer software, but which did not fit neatly into any of the preceding chapters. These two "loose ends" are the storage of MDBS directory information, the method for attaching or deleting language interface modules from the TI process, and certain limitations in the current MDBS implementation.

B. MDBS DIRECTORY INFORMATION

Certain information regarding the directory structure of MDBS is written into the code. The full pathnames of both the "home" directory and the data files directory (where users database schema and transaction files are stored) are hard-coded. The commdata.def file in the "version (greg)/COMMON" directory includes these two definitions:

```
#define DATA_AREA     "/u/mdbs/UserFiles/"
#define HOME          "/u/mdbs/"
```

The definition of the data files directory is repeated in the licommdata.def file found in the "/u/mdbs/version(greg)/CNTRL/TI/LangIF/include" directory.

The full pathname of the directory where the controller communication sockets are located is also hard-coded. The sndrcv.def file in the COMMON directory includes this definition:

```
#define PREFIX        "u/mdbs/Sockets"
```

Any change to this directory structure, including a change to the name of the root directory ("/u") to which this file system is mounted, would require changes to the above listed files. The makefiles are written using relative path names, so minor changes should not affect the integrity of these files.

C. ADDITION/DELETION OF LANGUAGE INTERFACE MODULES

MDBS has flexible architecture in terms of both its hardware and its software. Different versions of the software include different mixtures of non-kernel language interface modules. The purpose of this section is to identify the critical linkages between the kernel and non-kernel interfaces contained within the test interface (TI) code.

First, for any of the non-kernel language interfaces to be included, the language interface flag (LangIF_Flag) must be visible to the compiler. This means that the "#define LangIF_Flag" statement in the file "Flags.def" located in the TI directory must not be commented out.

Second, there must be function call to initialize the specific non-kernel language interface. This is accomplished by loading the schema for the non-kernel model. This call (*e.g.*, creat_rel_db_list) should be located around line 90 in the ti.c file.

Finally, it is necessary to add a menu choice and a call to the main procedure for the desired language interface. This code should be placed within the while loop following the function call to initialize the interface.

Once these changes are made and ti.exe is recompiled (which may require the modification of one or more makefiles), the new language interface should be available. Assuming the new language interface is properly written, the thread of execution in the TI process may now be switched to the new language interface. Removing a language interface(s) is accomplished by reversing the above steps. For more information on the design of a non-kernel language interface, see [Bourgeois, 1993, p. 29 - 36].

VII. CONCLUSIONS

A. AN INTRODUCTION

The Multibackend Database Supercomputer offers elegant, practical solutions to the four most pressing large database problems facing government and industry. MDBS allows data sharing, resource consolidation, scalable performance, and capacity growth.

B. WHAT HAS BEEN ACCOMPLISHED

In light of its promise, there is a need for a theory of MDBS system software portability. Such a theory identifies specific issues and problem areas for MDBS portability. It also increases the understanding of portability issues for parallel databases in general.

In this thesis, such a theory of system software portability has been developed. Portability has been addressed from the standpoint of hardware issues, operating system software issues, and communication issues. General types of problems as well as specific examples of problems and solutions have been presented. It is hoped that this information will facilitate the porting of MDBS to other platforms and stimulate interest in and development of the concepts embodied in MDBS. We have attempted to incorporate the knowledge gained through seven months of studying and experimenting with MDBS into this thesis so that the learning curve for future researchers and programmers will be eased. We have also sought to point out limitations in the current implementation of MDBS which can be corrected or enhanced.

C. TOPICS FOR FURTHER RESEARCH

MDBS is a large and complicated system. It has developed significantly over the last dozen years, but there is still more which can be done. This section discusses topics for further research.

One area for research involves improving the user and database capabilities of MDBS. Currently, MDBS can only support a single user and one database at a time. MDBS needs to be able to support multiple users and multiple databases simultaneously. Some work has already been done in this regard. The overall design of MDBS supports multiple users and databases (*e.g.* the meta-data and base-data disk organization). Many of the data structures related to users and databases are written so as to be easily expandable. This work should be continued.

Another research area is the expansion of the cross-model access capabilities of MDBS. Currently, the concept has been proved through the creation of a relational-to-hierarchical cross-model access module. Practical cross-model accessibility is possibly the most important potential contribution of MDBS. This capability should be expanded and studied further.

A final area for further research is the development of a more intuitive and efficient user interface for MDBS. The current command-line interface, which presents the user with a long series of sometimes confusing choices, is not a suitable interface for something with the power and flexibility of MDBS. The user interface issue needs to be thoroughly examined. The goal should be nothing less than redesigning the way in which MDBS interacts with the user.

D. IMPLEMENTATION ISSUES

There is another class of issues which may not involve pure research but which are still of great significance to the development of MDBS and its use by students at the Naval Postgraduate School. This section presents four such issues.

First, the multi-model and multi-language capabilities which have been demonstrated and used for instruction should be implemented on the multi-backend version of the system. The network, hierarchical, and object-oriented model-language interfaces are currently implemented only on the single machine version of MDBS. Some effort has already been expende o begin moving these interfaces to the multi-backend version. This work should be continued, as the implementation of a single multi-model, multi-language, and multi-backend system will benefit students, researchers, and staff workers alike.

Second, a review of the value used for TrackSize should be made. It can be assumed from its name that this value was once the physical track size of the base-data (record) disk. This is not now true, nor was it true on the previous hardware platform. The value of TrackSize is used in the determination of the value of MSGLEN (message length) in msg.def. It also figures in the calculation for the maximum number of fragments into which an inter-machine message may be broken. Interestingly, the value of MSGLEN in licommdata.def is different from the one in meta.def. These inconsistencies need to be resolved.

Third, a review of the number and distribution of header files should be made. Currently, definitions are dispersed over a large number of directories. These could be combined into fewer files, each with a common purpose. For example, it would be useful to have those definitions pertaining to communications parameters in a single file.

The fourth issue is a small one. That portion of the TI process which handles the kernel model and language, does not automatically generate descriptor (*.d) and template (*.t) files for the user. The other language interface modules (e.g. the relational model and language interface) generate the descriptor and template files for the user based upon input from a description of the database schema. The user of the kernel (ABDL) interface must create these files off-line and add them to the system manually. A mechanism for creating these critical files should be added to the ABDL interface.

E. RECOMMENDATIONS

MDBS is a research tool, not a commercial product. It has been the subject of work by many different students and researchers over many years. As a result, there are a few shortcomings in its implementation. This section makes some recommendations for strengthening the code.

It is recommended that more structure be added to the coding of MDBS. The current implementation of MDBS uses the pre-ANSI (*i.e.*, K & R) C language. This permits an overuse of global variables and pointers which severely complicates the code and makes maintenance or modification difficult. It is recommended that MDBS be rewritten in a more structured language (including, possibly, ANSI C). This would allow greater error checking at compile-time, improve portability, and make the system much easier to understand.

It is recommended that a higher-quality compiler be procured for use by the researchers and programmers. The current command-line compiler has noticeable inconsistencies and provides little compile-time error checking.

It is recommended that the commenting of MDBS code be improved. Currently, large areas of the existing code are not well commented. Until recently, many of the files did not even include the name of the file as a comment. Lack of comments increases the difficulty in reading and understanding the code.

It is recommended that two sections of the code should be re-written to improve their portability. The first of these is ack.c (located in the COMMON directory). This file was written after most of the other MDBS code and contains several strongly implementation dependent functions. The second section of code which should be rewritten is the dbl_template() function in the dbl.c file. The nature of the dependencies are detailed in Chapter IV, section C.

Finally, it is recommended that a staff programmer be permanently assigned to assist with the MDBS project. The addition of a programmer/assistant would benefit the project, the researchers, and the students. This individual could serve as a source for advice on programming questions, as an expert trouble-shooter, and as the long-term "corporate knowledge" for the project's implementation details.

APPENDIX A. SYSTEM COMPARISON

A. HARDWARE

This following is a summary of the hardware comprising the old and new platforms. Note that the same standard Ethernet communications network is used with both systems.

1. Old hardware

a. Controller: ISI-V model V24S workstation (quantity one)

Table 1: OLD CONTROLLER HARDWARE

Item	Detail
host name	db8
central processing unit	Motorola 68020 (16 MHz)
physical ram	4 MB
fixed disk drive(s)	two Control Data CDC 86L ESDI drives, each with 101 MB capacity (MDBS utilizes a 100 MB partition mounted as "/u").
tape drive(s)	one 1/2" reel, one 1/4" cartridge
communications backplane(s)	two

b. Backend: ISI-V model V24S workstation (quantity six)

Table 2: OLD BACKEND HARDWARE

Item	Detail
host names	db3, db4, db5, db6, db7, db9

Table 2: OLD BACKEND HARDWARE

Item	Detail
central processing unit	Motorola 68020 (16 MHz)
physical ram	4 MB
fixed disk drive(s)	two Control Data CDC 86L ESDI drives, each with 101 MB capacity (MDBS utilizes one as a meta disk (raw device) and one for program code storage (@2 MB, mounted as "/u"), and one Control Data CDC Swallow ESDI drive with 399 MB capacity used by MDBS as a data disk (raw device)
communications backplane(s)	one

2. New Hardware

a. Controller: Sun model 4/110 workstation (quantity 1)

Table 3: NEW CONTROLLER HARDWARE

Item	Detail
host name	db11
central processing unit	Sun Sparc (RISC)
physical ram	8 MB
fixed disk drive(s)	one Micropolis 1558 with 373 MB capacity (MDBS utilizes @100MB on a partition mounted as "/u".
tape drive(s)	none (backup accomplished via network or portable tape unit)
communications backplane(s)	two

b. Backend: Sun model 4/280 workstation (quantity two)

Table 4: NEW BACKEND HARDWARE

Item	Detail
host names	db12, db13
central processing unit	Sun Sparc (RISC)
physical ram	16 MB
fixed disk drive(s)	one Hitachi DK 815-10 SCSI drive with 600+ MB capacity (MDBS uses only @2MB on file system mounted as "/u" for program storage) and two Quantum ProDrive 105S SCSI drives each with 100MB capacity (MDBS uses one as a meta disk and one as a data disk - both are unmounted raw devices)
tape drive(s)	one 1/4" cartridge drive
communications backplane(s)	one

B. SOFTWARE

The following is a summary of operating system and compiler software on the old and new systems.

1. Old Software

Table 5: OLD SYSTEM SOFTWARE

Item	Description
Operating System	ISIV version of BSD4.3 UNIX
Compiler	Standard C compiler (cc)

2. New Software

Table 6: New System Software

Item	Description
Operating System	SunOS 4.1.1
Compiler	Standard C compiler (cc)

APPENDIX B. AIDS TO DEBUGGING

A. AN INTRODUCTION

The suggestions offered in this appendix are meant to be of assistance to persons modifying or trouble-shooting the MDBS system software. These hints are loosely divided into six classes.

B. USING DEBUGGING FLAGS

A very useful and flexible set of debugging routines have been built into the MDBS code. These debugging routines, most of which print (printf()) useful information to the terminal or trace files, are included amongst the working code in conditional compilation constructs. These take the form:

```
#ifdef identifier
   (debugging code)
#endif
```

The *identifier* is the name of the debugging flag. The debugging code within the construct is only compiled into the executable file if the identifier is defined in the header file specified by the appropriate makefile. These header files are all named flags.def. One is located in each of the process directories in which compilation takes place (ten in all). The flag names are never actually removed from the flags.def files. They are commented out to prevent inclusion of the debugging code. Each flags.def file contains a brief explanation of the purpose of the flags contained therein. A number of new debugging statements which may benefit persons porting MDBS were added during this porting project.

C. USING TRACE FILES

Trace files are text files which contain the output of processes which do not write to the terminal. Careful analysis of these files is the best way to trouble-shoot MDBS or learn exactly how it performs most of its functions. The start.cntrl (controller) and run.be (backend) shell programs determine whether a process writes to the terminal or to a trace file. Normally, only the TI process writes to the terminal. The others write to trace files, all of which end in *.tr. The controller trace files are stored in the version (*e.g.*, greg)/run/trace directory on the controller and the be.version (*e.g.*, greg) directory on each backend machine.

The amount of information which is output to the trace file is determined by the setting of the debugging flags for that process. Most of the information in a trace file is readily comprehensible once the codes are understood. These codes are contained in the msg.def header file.

Often a clearly labeled error message will direct the reader to the source of a problem. Other times it is necessary to study all of the traces together while looking for an abnormal pattern. Depicting the message flow on a copy of the communication channel map (Figure 8 in Chapter V) can be a tremendous help on these occasions. This is especially true where the death of one process causes some of the other connected processes to die. Abnormal termination of one MDBS process often causes a "domino" effect on the other processes.

D. CHECKING SYSTEM STARTUP

A few simple observations can help locate software problems which occur during system start-up. Check the Sockets directory to see if all of the processes are creating sockets. A new socket is created for every process on every run of the system. Check the trace file directories to see if all of the processes are creating their trace files. These are also deleted and created on each new run. Check to see if a process identification file (.pid file) has been created in the run directory on each machine. If any of these are missing, the system initialization is not progressing normally.

Once the system appears to be up and running, execute a process status (ps) on both the controller and the backends. Check to see that all twelve MDBS processes (six on the controller, six on the backend) are running. Remember that there is a short delay before the get processes are started.

E. MAINTAINING A HISTORY

The generation and retention of detailed information about every modification and the system's response is strongly recommended. Use of the UNIX script utility allows the trouble-shooter to capture screen output (such as the TI process output) or keep a permanent record of the contents of the trace files from any given run. Scripting the results of the execution of the make_results shell program (located in the version/bin directory) produces a record of all of the compiler's messages from the last make. Scripting system tests provides a permanent reference. Scripting the trace files for detailed analysis is also helpful.Something similar to the tp8.stan and tp13.stan shell programs (located with the trace files) may be helpful here.

F. UNIX DEBUGGING TOOLS

The standard UNIX trouble-shooting tools (*e.g.*, dbx, lint) were of limited assistance with MDBS problems. Most of these tools are useful for trouble-shooting a single process, but lack the facilities for running twelve processes at once. The dbx tool may be useful where one particular process is causing a serious problem such as a segmentation error.

G. PROGRAMMING TIPS

When making changes to code which is local to one process (those not contained in a COMMON directory), always execute a make (mk*) in that directory before trying a system-wide makeall. The system-wide make takes from as little as twelve minutes to as much as thirty-seven minutes, depending on how much of the language-interface code involved. This is a long time to wait only to discover a minor syntax error.

Whenever adding new debugging code, always immediately follow the printf() statement with an fflush(stdout) statement. This ensures that the output of the printf() statement goes immediately to the appropriate trace file, rather than languishing in the buffer where it could be lost in case of process termination.

Lastly, it is often difficult to determine the time sequence of operations performed in different processes. Adding the time(0) call to a printf() statement can provide useful clues in the trace files. The time(0) call returns the system time (in whole seconds) as an integer.

APPENDIX C. CONTROLLER DIRECTORY AND FILE STRUCTURE

The following is a listing of the important directories and files which make up the controller on MDBS.

A. THE "mdbs" DIRECTORY

This is the "mdbs" directory on the controller. All the other directories descend from this one. It also contains the temporary files (*.pid) used to store process identification numbers for the controller processes. These files are deleted and then recreated on each run. The path to this directory (/u/mdbs/) is hard-coded into the MDBS software as "HOME" in the commdata.def file.

```
total 75
drwxrwxr-x 14 mdbs          1024 Jul  6 13:05 ./
drwxr-xr-x 13 root           512 Sep 18  1992 ../
-rw-rw-r--  1 mdbs          4387 May 29 09:59 .alias
-rw-rw-r--  1 mdbs             5 Jul  6 13:05 .cget.exe.pid
-rw-rw-r--  1 mdbs             5 Jul  6 13:05 .cput.exe.pid
-rw-rw-r--  1 mdbs           242 Apr 26 12:42 .cshrc
-rw-rw-r--  1 mdbs             5 Jul  6 13:05 .iig.exe.pid
-rw-rw-r--  1 mdbs             5 Jul  6 13:05 .pp.exe.pid
-rw-rw-r--  1 mdbs             5 Jul  6 13:05 .reqp.exe.pid
-rw-r--r--  1 mdbs           114 Mar  8 12:20 .rhosts
-rw-rw-r--  1 mdbs          1189 Feb  4 14:44 .rhosts.bak
-rw-rw-r--  1 mdbs             5 Jul  6 13:05 .ti.exe.pid
drwxr-xr-x  2 mdbs           512 Jul  6 13:05 Sockets/
drwxr-xr-x 10 mdbs          2048 Jul  6 13:05 UserFiles/
drwxrwxr-x  2 mdbs           512 Apr 30 12:07 bin/
drwxr-xr-x  7 mdbs          1536 May 10 19:42 greg/
```

B. THE "Sockets" DIRECTORY

The "sockets" directory contains the six sockets used for inter-process-communication in the controller. The G_PCLC socket is used by the CGET process. The P_PCLC socket is used by the CPUT process. All of these sockets are deleted and recreated for every run.

```
Sockets:
total 2
drwxr-xr-x  2 mdbs           512 Jul  6 13:05 ./
```

```
drwxrwxr-x 14 mdbs          1024 Jul  6 13:05 ../
srwxrwxrwx  1 mdbs             0 Jul  6 13:05 G_PCLC=
srwxrwxrwx  1 mdbs             0 Jul  6 13:05 IIG=
srwxrwxrwx  1 mdbs             0 Jul  6 13:05 PP=
srwxrwxrwx  1 mdbs             0 Jul  6 13:05 P_PCLC=
srwxrwxrwx  1 mdbs             0 Jul  6 13:05 REQP=
srwxrwxrwx  1 mdbs             0 Jul  6 13:05 TI=
```

C. THE "UserFiles" DIRECTORY

The "UserFiles" directory contains the mass load files (*.r) and the controller's copy of the descriptor (*.d) and template (*.t) files for each database used with the system. It also contains the schema information files (*db) and stored transaction request files (*req) for each database. The one transaction file containing the letters RTH (*e.g.* SQDRTHreq) is part of the "relational-to-hierarchical" cross-model accessing capability of MDBS). This controller is set up for use with three databases (*i.e.* COURSE, SALES, and SQD). For details on how these files are used, see [Bourgeois, 1993]. The path to this directory (/u/mdbs/UserFiles) is hard-coded into the MDBS software as "DATA_AREA" in the commdata.def. file. The directories under this one (abdm, daplex, hierarchical, etc.) hold duplicate copies of these same database files for each model: Their contents are not listed here for the sake of brevity.

```
UserFiles:
total 105
drwxr-xr-x 10 mdbs          2048 Jul  6 13:05 ./
drwxrwxr-x 14 mdbs          1024 Jul  6 13:05 ../
-rw-r--r--  1 mdbs            13 Jun  2  1992 .pw
-rw-rw-r--  1 mdbs            51 Feb 25 21:56 COURSE.d
-rw-r--r--  1 mdbs           172 Feb 25 21:25 COURSE.r
-rw-rw-r--  1 mdbs           126 Apr  9 12:11 COURSE.t
-rw-r--r--  1 mdbs           316 Oct 19  1992 COURSEsqldb
-rw-r--r--  1 mdbs           613 Oct 19  1992 COURSEsqlreq
-rw-r--r--  1 mdbs           289 Oct 13  1992 SALESreq
-rw-r--r--  1 mdbs           120 Oct 13  1992 SALES.d
-rw-r--r--  1 mdbs           263 Oct 13  1992 SALES.r
-rw-r--r--  1 mdbs           121 Oct 13  1992 SALES.t
-rw-rw-r--  1 mdbs            43 Mar 31 15:16 SQD.d
-rw-rw-r--  1 mdbs           140 Mar 31 15:16 SQD.t
-rw-rw-r--  1 mdbs           828 Feb 10 12:35 SQDRTHreq
-rw-rw-r--  1 mdbs           813 Feb 10 12:36 SQDreq
drwxrwxr-x  2 mdbs           512 Oct  6  1992 abdm/
drwxrwxr-x  2 mdbs           512 Oct  6  1992 daplex/
```

```
drwxrwxr-x  2 mdbs          512 Oct  6  1992 hierarchical/
drwxrwxr-x  2 mdbs          512 Oct  6  1992 network/
drwxrwxr-x  2 mdbs         1024 Oct 24  1992 relational/
drwxrwxr-x  2 mdbs          512 Oct  6  1992 s_and_f_files/
```

D. THE "bin" DIRECTORY

This is the uppermost of two "bin" directories in MDBS (the other is a subdirectory of the "version" directory). This directory is used to store utility files, none of which are used directly by MDBS. The "cpydisks" script was once used for distributing files. The three scripts beginning with 'z' were used for zeroing meta and data disks. Both functions are now accomplished in other ways.

```
bin:
total 134
drwxrwxr-x  2 mdbs          512 Apr 30 12:07 ./
drwxrwxr-x 14 mdbs         1024 Jul  6 13:05 ../
-rwxr-xr-x  1 mdbs          122 Jan 14  1989 .z*
-rwxr-xr-x  1 mdbs           84 Sep  1  1989 cpydisks*
-rwxr-xr-x  1 mdbs           48 Jun 14  1989 z*
-rwxr-xr-x  1 mdbs          108 Sep  1  1989 zip*
```

E. THE VERSION (*e.g.*, "greg") DIRECTORY

The "version" directory serves as the top-level directory for each version of the software on the system. Each version of MDBS software has a unique copy of this directory and all of its subdirectories. The current version of the software is called "greg", hence the name of this directory. For details on version control, see [Meeks, 1993].

```
greg:
total 1151
drwxr-xr-x  7 mdbs         1536 May 10 19:42 ./
drwxrwxr-x 14 mdbs         1024 Jul  6 13:05 ../
drwxr-xr-x  8 mdbs          512 Jun  2 12:50 BE/
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 CNTRL/
drwxr-xr-x  2 mdbs         1024 Apr 23 09:20 COMMON/
drwxr-xr-x  2 mdbs         1024 Jun  2 11:31 bin/
-rw-rw-r--  1 mdbs         1511 Oct 16  1992 d_u
-rw-rw-r--  1 mdbs        33666 May 10 19:26 du_a
drwxr-xr-x  7 mdbs         1536 Jul  6 13:07 run/
```

F. THE "version/bin" DIRECTORY

This is the second "bin" directory in the MDBS controller. This one contains many important files and utilities. The makefile in this directory is used to control compilation and linking of all twelve (six controller, six backend) MDBS processes. Executing "make clean" followed by "makeall" from this directory will remove all object files from the twelve process directories and then create and distribute the new executables. Note that this process does not remove the object files making up the "TI" process which are located below the "src" directory in the language interface modules. These files must be removed manually. The shell script "make_results" displays the compiler and linker messages from the last make. The two source files (main.c, configure.h) for the "main" executable, which automates the running of MDBS, are located here. The working copy of the main executable is copied to the "run" directory. The zero executable is copied here after being created in the "DIO" directory. Zero handles the initialization of the backend data and meta disks. It must be manually copied to the "bin" directory on each backend machine. Cpcount is an executable which copies a specified number of bytes from one file to another. Rectag is an executable utility for manipulating the data disk on a backend machine. It must be copied to the backend "bin" directory. The cpydisks shell script was once used to distribute the executable processes to the backend machines. It has been superseded by newer code written into the "main" executable. The stop.cmd shell script is an older version of the files used to stop MDBS processes. The newer ones are located in the "run" directory.

```
greg/bin:
total 422
drwxr-xr-x  2 mdbs         1024 Jun  2 11:31 ./
drwxr-xr-x  7 mdbs         1536 May 10 19:42 ../
-rw-rw-r--  1 mdbs          122 Feb 25 12:54 Makefile
-rwxrwxr-x  1 mdbs        37152 Feb  5 13:40 configure*
-rw-rw-r--  1 mdbs         5673 May 19 10:39 configure.h
-rwxr-xr-x  1 mdbs        17948 Jun  2  1992 constants*
-rwxr-xr-x  1 mdbs        26479 Jun  2  1992 cpcount*
-rwxr-xr-x  1 mdbs           84 Jun  2  1992 cpydisks*
-rwxrwxr-x  1 mdbs        70533 Jul  2 15:23 disp*
-rwxrwxr-x  1 mdbs        37611 May 19 10:40 main*
-rw-rw-r--  1 mdbs        33550 May 19 10:35 main.c
-rw-rw-r--  1 mdbs        14194 May 19 10:40 main.o
```

```
-rwxr--r--  1 mdbs          932 Apr 30 12:05 make_results*
-rwxr-xr-x  1 mdbs          557 Apr  9 11:42 makeall*
-rw-r--r--  1 mdbs          664 Apr  9 12:14 makefile
-rwxr-xr-x  1 mdbs        26590 Jun  2  1992 rectag*
-rwxr-xr-x  1 mdbs          306 Feb  5 13:44 stop.cmd*
-rw-rw-r--  1 mdbs         1984 Apr 30 12:08 temp.txt
-rwxr--r--  1 mdbs        26439 Jun  2  1992 zero*
```

G. THE "run" DIRECTORY

This is the directory from which MDBS is normally run. It contains the executables and scripts necessary to control the orderly generation of the system. The "main" executable is the program which controls all of the others. It calls the zero command in each backend using the zero.db* scripts. It calls run.be on each backend to start the backend processes. The master.run.be.file is a master copy of the files located on the backend machines. Main also calls start.cntrl to start the controller processes. Ultimately, it calls the stop.db* scripts to stop the processes when shutdown is signalled. This directory also contains numerous temporary files created by MDBS. The *.dbl files are database listing files. Each contains the names of the databases that exist for each model. The *dbs.dat (database data) files hold schema information about each database. The information in the *dbs.dat files are used to generate a catalog file (*.cat) for each database. The .qry_file and .TransFile temporary files hold information about the latest querry transactions. The .config.db file stores information (*i.e.*, machine host names) for the most recent MDBS configuration. The "trace" subdirectory contains trace (*.tr) files generated on the latest run of MDBS. Each trace file contains the output from the controller process of the same name. The backend process trace files may be found on each backend machine.

```
greg/run:
total 243
drwxr-xr-x  7 mdbs         1536 Jul  6 13:07 ./
drwxr-xr-x  7 mdbs         1536 May 10 19:42 ../
-rw-rw-r--  1 mdbs          266 Apr 13 13:31 .COURSE.cat
-rw-rw-r--  1 mdbs          270 Apr  9 12:13 .DTH.cat
-rw-rw-r--  1 mdbs          156 Apr  9 12:13 .SQD.cat
-rw-r--r--  1 mdbs          353 Apr  9 12:13 .Syntax
-rw-rw-r--  1 mdbs          929 Apr 23 11:28 .TransFile
-rw-rw-r--  1 mdbs            9 Apr 26 15:27 .config.db
-rw-r--r--  1 mdbs           10 Apr  9 12:13 .curr_file
```

```
-rwxr-xr-x  1 mdbs          0 Apr  9 12:13 .dapdbs.dat*
-rw-------  1 mdbs         19 Apr  9 12:13 .exe.awk
-rw-rw-r--  1 mdbs          4 Apr  9 12:13 .hie.dbl
-rw-rw-r--  1 mdbs        154 Apr  9 12:13 .hiedbs.dat
-rw-rw-r--  1 mdbs         13 Apr  9 12:13 .net.dbl
-rw-r--r--  1 mdbs        726 Apr  9 12:13 .netdbs.dat
-rw-rw-r--  1 mdbs          0 Apr 23 11:28 .output
-rw-rw-r--  1 mdbs        728 Apr 23 11:29 .qry_file
-rw-rw-r--  1 mdbs        698 Apr  9 12:13 .reldbs.dat
-rw-rw-r--  1 mdbs         16 Apr 13 13:31 .sql.dbl
-rwxrwxr-x  1 mdbs      37611 May 19 10:40 main*
-rwxr--r--  1 mdbs        436 Nov  6  1992 master.run.be*
-rwxr--r--  1 mdbs        426 Mar  4 13:37 start.cntrl*
-rwxr-xr-x  1 mdbs         76 Feb 23 12:12 stop.db1*
-rwxr-xr-x  1 mdbs         76 Feb 23 12:12 stop.db2*
-rwxr-xr-x  1 mdbs         76 Nov  6  1992 stop.db3*
-rwxr-xr-x  1 mdbs         76 Nov  6  1992 stop.db4*
-rwxr-xr-x  1 mdbs         76 Nov  6  1992 stop.db5*
-rwxr-xr-x  1 mdbs         76 Nov  6  1992 stop.db6*
-rwxr-xr-x  1 mdbs         76 Nov  6  1992 stop.db7*
-rwxr-xr-x  1 mdbs        314 Mar  4 13:59 stop.db8*
-rwxr-xr-x  1 mdbs         76 Nov  6  1992 stop.db9*
drwxrwxr-x  2 mdbs        512 Jul 15 10:04 trace/
-rwxr--r--  1 mdbs        230 Feb 23 13:31 zero.db1*
-rwxr--r--  1 mdbs        230 Feb 23 13:31 zero.db2*
-rwxr--r--  1 mdbs        230 Feb 23 13:31 zero.db3*
-rwxr--r--  1 mdbs        230 Feb 23 13:31 zero.db4*
-rwxr--r--  1 mdbs        309 Feb  5 13:30 zero.db5*
-rwxr--r--  1 mdbs        279 Feb 23 13:32 zero.db6*
-rwxr--r--  1 mdbs        230 Feb 23 13:32 zero.db7*
-rwxr--r--  1 mdbs        719 Apr  7 11:17 zero.db8*
-rwxr--r--  1 mdbs        279 Feb 23 13:32 zero.db9*
```

H. THE "version/COMMON" DIRECTORY

The COMMON directory under the version directory contains source code common to both controller and backend processes. Much of the code dealing with inter-process and inter-machine communications is located here. Numerous hardware and network specific definitions are also contained in the header files located here.

```
greg/COMMON:
total 241
drwxr-xr-x  2 mdbs       1024 Apr 23 09:20 ./
drwxr-xr-x  7 mdbs       1536 May 10 19:42 ../
-rw-r--r--  1 mdbs      16679 Jun  2  1992 ack.c
-rw-r--r--  1 mdbs       1279 Jun  2  1992 ack.dcl
-rw-r--r--  1 mdbs        320 Jun  2  1992 ack.def
-rw-r--r--  1 mdbs         48 Jun  2  1992 beno.dcl
```

```
-rw-r--r--  1 mdbs          69 Jun  2  1992 beno.def
-rw-r--r--  1 mdbs        8661 Jun  2  1992 cb.c
-rw-r--r--  1 mdbs        6706 Jun  2  1992 comio.c
-rw-r--r--  1 mdbs       13788 Jun  2  1992 commdata.def
-rw-r--r--  1 mdbs         615 Jun  2  1992 commsg.c
-rw-r--r--  1 mdbs         548 Jun  2  1992 dblgeneral.c
-rw-r--r--  1 mdbs        7774 Jun  2  1992 dbtmpmod.c
-rw-r--r--  1 mdbs        3836 Jun  2  1992 errormsg.c
-rw-r--r--  1 mdbs        9360 Jun  2  1992 generals.c
-rw-r--r--  1 mdbs        1133 Jun  2  1992 msend.c
-rw-r--r--  1 mdbs          63 Jun  2  1992 msg.dcl
-rw-r--r--  1 mdbs        6958 Jun  2  1992 msg.def
-rw-r--r--  1 mdbs         156 Jun  2  1992 msg.ext
-rw-r--r--  1 mdbs         555 Jun  2  1992 newdb.c
-rw-r--r--  1 mdbs        1694 Jun  2  1992 newtmpl.c
-rw-r--r--  1 mdbs       28116 Jun  2  1992 pcl.c
-rw-r--r--  1 mdbs         899 Apr  2 11:35 pcl.def
-rw-r--r--  1 mdbs         590 Jun  2  1992 select.c
-rw-r--r--  1 mdbs         369 Jun  2  1992 setbeno.c
-rw-r--r--  1 mdbs         350 Jun  2  1992 setnobes.c
-rw-r--r--  1 mdbs       27857 Jun 14  1992 sndrcv.c
-rw-r--r--  1 mdbs        1734 Jun  2  1992 sndrcv.dcl
-rw-r--r--  1 mdbs         296 Jun  2  1992 sndrcv.def
-rw-r--r--  1 mdbs         946 Jun  2  1992 sndrcv.ext
-rw-r--r--  1 mdbs         173 Jun  2  1992 tmpl.dcl
-rw-r--r--  1 mdbs         804 Jun  2  1992 tmpl.def
-rw-r--r--  1 mdbs         188 Jun  2  1992 tmpl.ext
-rw-r--r--  1 mdbs        7569 Jun  2  1992 utilities.c
-rw-r--r--  1 mdbs        1592 Jun  2  1992 waitmsg.c
```

I. THE "BE" DIRECTORY

The BE directory is the top-level directory for all of the backend source code and executables. Under the current approach, all code (including backend code) is compiled on the controller and then copied to its appropriate destination. A copy of the backend executables are kept in this directory. The six subdirectories under this directory hold source code, object code, and makefiles for each backend process.

```
greg/BE:
total 1864
drwxr-xr-x  8 mdbs         512 Jun  2 12:50 ./
drwxr-xr-x  7 mdbs        1536 May 10 19:42 ../
drwxr-xr-x  3 mdbs         512 Jun  2 11:58 BCOM/
drwxr-xr-x  3 mdbs         512 Jun  2 11:47 CC/
drwxr-xr-x  3 mdbs         512 Oct 25  1992 COMMON/
drwxr-xr-x  3 mdbs         512 Jun  2 11:49 DIO/
drwxr-xr-x  3 mdbs        1024 Jun  2 11:51 DM/
```

```
drwxr-xr-x  3 mdbs         1024 Jun  2 12:00 RECP/
-rwxrwxr-x  1 mdbs        91847 Jul  6 13:01 bget.exe*
-rwxrwxr-x  1 mdbs        91847 Jul  6 13:01 bput.exe*
-rwxrwxr-x  1 mdbs       104564 Jul  2 14:37 cc.exe*
-rwxrwxr-x  1 mdbs        55518 Jul  2 14:36 dio.exe*
-rwxrwxr-x  1 mdbs       126485 Jul  2 15:23 dirman.exe*
-rwxrwxr-x  1 mdbs       150366 Jul  2 14:50 recp.exe*
```

J. THE "BE/COMMON" DIRECTORY

This directorv holds code which is shared by two or more backend processes. Its "Object" subdirectory is used to store a significant amount of object code used by backend processes.

```
greg/BE/COMMON:
total 7
drwxr-xr-x  3 mdbs          512 Oct 25  1992 ./
drwxr-xr-x  8 mdbs          512 Jun  2 12:50 ../
drwxr-xr-x  2 mdbs          512 Jun  2 11:59 Object/
-rw-r--r--  1 mdbs         3349 Jun  2  1992 tmplsr.c
```

K. THE "BE/BCOM" DIRECTORY

This directory contains the source code needed to compile the backend get (BGET) and put (BPUT) processes. Flags.def contains flags whose setting determine what, if any, debugging code will be compiled into the executable files. The mk* script can be used to recompile just the code in this subdirectory. System-wide recompilation is controlled from the "version/bin" directory. The make_result file contains the compiler's comments (errors, warnings, etc.) about the most recent compilation. The "Object" subdirectory contains the object code generated from this source code as well as the makefile and shell scripts for copying executables to their proper location.

```
greg/BE/BCOM:
total 14
drwxr-xr-x  3 mdbs          512 Jun  2 11:58 ./
drwxr-xr-x  8 mdbs          512 Jun  2 12:50 ../
drwxr-xr-x  2 mdbs          512 Jul  6 13:01 Object/
-rw-r--r--  1 mdbs         3281 Jun  2  1992 bget.c
-rw-r--r--  1 mdbs         1612 Jun  2  1992 bput.c
-rw-r--r--  1 mdbs           33 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs          841 Jul  6 12:59 flags.def
-rw-rw-r--  1 mdbs          394 Jul  6 13:01 make_result
```

```
-rwxr--r--   1 mdbs          106 Jun  2  1992 mk*
```

L. THE "BE/CC" DIRECTORY

This directory contains the source code needed to make the executable for the concurrency control (CC) process. The makefiles and shell scripts function like those in the "BCOM" directory.

```
greg/BE/CC:
total 202
drwxr-xr-x   3 mdbs          512 Jun  2 11:47 ./
drwxr-xr-x   8 mdbs          512 Jun  2 12:50 ../
-rw-r--r--   1 mdbs          504 Jun  2  1992 .fixed
drwxr-xr-x   2 mdbs         1024 Jul  2 14:38 Object/
-rw-r--r--   1 mdbs        21349 Jun  2  1992 atut.c
-rw-r--r--   1 mdbs          371 Jun  2  1992 cc.dcl
-rw-r--r--   1 mdbs         6719 Jun  2  1992 cc.def
-rw-r--r--   1 mdbs          443 Jun  2  1992 cc.ext
-rw-r--r--   1 mdbs        16851 Jun  2  1992 cccs.c
-rw-r--r--   1 mdbs        17883 Jun  2  1992 ccds.c
-rw-r--r--   1 mdbs        10408 Jun  2  1992 ccmain.c
-rw-r--r--   1 mdbs        16869 Jun  2  1992 ccrp.c
-rw-r--r--   1 mdbs        13063 Jun  2  1992 ccsr.c
-rw-r--r--   1 mdbs          680 Jun  2  1992 cinit.c
-rw-r--r--   1 mdbs        14478 Jun  2  1992 ctut.c
-rw-r--r--   1 mdbs           31 Jun  2  1992 dblocal.def
-rw-r--r--   1 mdbs          824 Jul  2 14:32 flags.def
-rw-rw-r--   1 mdbs          848 Jul  2 14:37 make_result
-rw-r--r--   1 mdbs         7214 Jun  2  1992 mallocs.c
-rwxr--r--   1 mdbs          107 Jun  2  1992 mk*
-rw-r--r--   1 mdbs        24214 Jun  2  1992 tuat.c
-rw-r--r--   1 mdbs        15258 Jun  2  1992 tuct.c
-rw-r--r--   1 mdbs        21221 Jun  2  1992 tudist.c
-rw-r--r--   1 mdbs         1156 Jun  2  1992 unixcinit.c
-rw-r--r--   1 mdbs          979 Jun  2  1992 update.c
```

M. THE "BE/DIO" DIRECTORY.

This directory holds the source code for dio.exe, the basis of the record disk input-output (DIO) process).

```
greg/BE/DIO:
total 36
drwxr-xr-x   3 mdbs          512 Jun  2 11:49 ./
drwxr-xr-x   8 mdbs          512 Jun  2 12:50 ../
drwxr-xr-x   2 mdbs          512 Jul  2 14:36 Object/
-rw-r--r--   1 mdbs         1610 Jun  2  1992 cpcount.c
```

```
-rw-r--r--  1 mdbs          33 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs       15993 Jun  2  1992 dio.c
-rw-r--r--  1 mdbs        2102 Jun  2  1992 dio.h
-rw-r--r--  1 mdbs         828 Jul  2 14:34 flags.def
-rw-rw-r--  1 mdbs         183 Jul  2 14:36 make_result
-rwxr--r--  1 mdbs         112 Jun  2  1992 mk*
-rw-r--r--  1 mdbs        4994 Jun  2  1992 rectag.c
-rw-r--r--  1 mdbs        1792 Jun  2  1992 zero.c
```

N. THE "BE/DM" DIRECTORY

This directory contains the source code for the directory management (DM) process.

```
greg/BE/DM:
total 289
drwxr-xr-x  3 mdbs        1024 Jun  2 11:51 ./
drwxr-xr-x  8 mdbs         512 Jun  2 12:50 ../
drwxr-xr-x  2 mdbs        1024 Jul  2 15:23 Object/
-rw-r--r--  1 mdbs        8329 Jun  2  1992 ag.c
-rw-r--r--  1 mdbs        7712 Jun  2  1992 atm.c
-rw-r--r--  1 mdbs        5455 Jun  2  1992 beno.c
-rw-r--r--  1 mdbs        4493 Jun  2  1992 cdtmbe.c
-rw-r--r--  1 mdbs        1055 Jun  2  1992 cdtmm.c
-rw-r--r--  1 mdbs        9648 Jun  2  1992 common.c
-rw-r--r--  1 mdbs        1688 Jun  2  1992 constants.c
-rw-r--r--  1 mdbs       14980 Jun  2  1992 cs.c
-rw-r--r--  1 mdbs        4503 Jun  2  1992 cs1resta.c
-rw-r--r--  1 mdbs        5999 Jun  2  1992 cs3rest.c
-rw-r--r--  1 mdbs        1562 Jun  2  1992 dbinit.c
-rw-r--r--  1 mdbs          31 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs        2369 Jun  2  1992 ddit.c
-rw-r--r--  1 mdbs        5262 Jun  2  1992 desc.c
-rw-r--r--  1 mdbs       14576 Jun  2  1992 didef.c
-rw-r--r--  1 mdbs       19862 Jun  2  1992 dirman.c
-rw-r--r--  1 mdbs         407 Jun  2  1992 dirman.dcl
-rw-r--r--  1 mdbs        5062 Jun  2  1992 dirman.def
-rw-r--r--  1 mdbs         421 Jun  2  1992 dirman.ext
-rw-r--r--  1 mdbs        7179 Jun  2  1992 disp.c
-rw-r--r--  1 mdbs        3247 Jun  2  1992 dmfree.c
-rw-r--r--  1 mdbs        1024 Jun  2  1992 dmnomore.c
-rw-r--r--  1 mdbs       13480 Jun  2  1992 dmsr.c
-rw-r--r--  1 mdbs         904 Jun  2  1992 dmupdfin.c
-rw-r--r--  1 mdbs       10234 Jun  2  1992 ds.c
-rw-r--r--  1 mdbs        3790 Jun  2  1992 dsdone.c
-rw-r--r--  1 mdbs         825 Jul  2 15:17 flags.def
-rw-rw-r--  1 mdbs        1491 Jul  2 15:23 make_result
-rw-r--r--  1 mdbs        5703 Jun  2  1992 mallocs.c
-rw-r--r--  1 mdbs       45200 Jun  2  1992 meta.c
-rw-r--r--  1 mdbs        9750 Jun  2  1992 meta.def
-rwxr--r--  1 mdbs         111 Jun  2  1992 mk*
```

```
-rw-r--r--  1 mdbs          6392 Jun  2  1992 newdesc.c
-rw-r--r--  1 mdbs          7425 Jun  2  1992 oldnew.c
-rw-r--r--  1 mdbs         16676 Jun  2  1992 rdtsave.c
-rw-r--r--  1 mdbs          1300 Jun  2  1992 rdtsort.c
-rw-r--r--  1 mdbs          4513 Jun  2  1992 tabledump.c
-rw-r--r--  1 mdbs         16447 Jun  2  1992 tu.c
```

O. THE "BE/RECP" DIRECTORY

This directory contains source code for the record processing (RECP) process.

```
greg/BE/RECP:
total 250
drwxr-xr-x  3 mdbs          1024 Jun  2 12:00 ./
drwxr-xr-x  8 mdbs           512 Jun  2 12:50 ../
drwxr-xr-x  2 mdbs          1024 Jul  2 14:50 Object/
-rw-r--r--  1 mdbs          9528 Jun  2  1992 allsto.c
-rw-r--r--  1 mdbs            48 Jun  2  1992 beno.dcl
-rw-r--r--  1 mdbs          5291 Jun  2  1992 chkqry.c
-rw-r--r--  1 mdbs           978 Jun  2  1992 chkwait.c
-rw-r--r--  1 mdbs           152 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs          1554 Jun  2  1992 delp.c
-rw-r--r--  1 mdbs          9466 Jun  2  1992 disks.c
-rw-r--r--  1 mdbs           145 Jun  2  1992 disks.dcl
-rw-r--r--  1 mdbs           773 Jun  2  1992 disks.def
-rw-r--r--  1 mdbs            63 Jun  2  1992 disks.ext
-rw-r--r--  1 mdbs          1463 Jun  2  1992 findrp.c
-rw-r--r--  1 mdbs           865 Jul  2 14:39 flags.def
-rw-r--r--  1 mdbs          3135 Jun  2  1992 insp.c
-rw-rw-r--  1 mdbs          1163 Jul  2 14:50 make_result
-rw-r--r--  1 mdbs          2219 Jun  2  1992 mallocs.c
-rwxr--r--  1 mdbs           111 Jun  2  1992 mk*
-rw-r--r--  1 mdbs          1132 Jun  2  1992 nomore.c
-rw-r--r--  1 mdbs          6632 Jun  2  1992 rbabs.c
-rw-r--r--  1 mdbs          6006 Jun  2  1992 rcreqs.c
-rw-r--r--  1 mdbs         19954 Jun  2  1992 recproc.c
-rw-r--r--  1 mdbs           257 Jun  2  1992 recproc.dcl
-rw-r--r--  1 mdbs          4875 Jun  2  1992 recproc.def
-rw-r--r--  1 mdbs           283 Jun  2  1992 recproc.ext
-rw-r--r--  1 mdbs         19690 Jun  2  1992 recpsr.c
-rw-r--r--  1 mdbs         10332 Jun  2  1992 retby.c
-rw-r--r--  1 mdbs         25912 Jun  2  1992 retcom.c
-rw-r--r--  1 mdbs         12334 Jun  2  1992 retp.c
-rw-r--r--  1 mdbs          1154 Jun  2  1992 rpcont.c
-rw-r--r--  1 mdbs          3377 Jun  2  1992 rpfree.c
-rw-r--r--  1 mdbs          4553 Jun  2  1992 streqs.c
-rw-r--r--  1 mdbs          1165 Jun  2  1992 unixdisks.c
-rw-r--r--  1 mdbs           433 Jun  2  1992 unixdisks.def
-rw-r--r--  1 mdbs         20414 Jun  2  1992 updp.c
-rw-r--r--  1 mdbs          4693 Jun  2  1992 wcreqs.c
```

P. THE "version/CNTRL" DIRECTORY

This is the top-level directory for controller source and object code. It contains the executables for each of the six controller processes (CGET, CPUT, IIG, PP, REQP, and TI). These executables are copied here by shell scripts after being created in the subdirectories bearing their names.

```
greg/CNTRL:
total 1784
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ./
drwxr-xr-x  7 mdbs         1536 May 10 19:42 ../
drwxr-xr-x  3 mdbs          512 Jun  2 11:34 CCOM/
drwxr-xr-x  3 mdbs          512 Oct 25  1992 COMMON/
drwxr-xr-x  3 mdbs          512 Jun 20  2:25 IIG/
drwxr-xr-x  3 mdbs          512 Jun  2 11:36 PP/
drwxr-xr-x  3 mdbs          512 Jun  2 11:38 REQP/
drwxr-xr-x  4 mdbs         1024 Jun  2 11:40 TI/
-rwxrwxr-x  1 mdbs        91811 Jun 20 13:09 cget.exe*
-rwxrwxr-x  1 mdbs        91811 Jun 20 13:09 cput.exe*
-rwxrwxr-x  1 mdbs        56680 Jun  2 12:48 iig.exe*
-rwxrwxr-x  1 mdbs        56971 Jun  2 12:48 pp.exe*
-rwxrwxr-x  1 mdbs        84382 Jun  2 12:48 reqp.exe*
-rwxrwxr-x  1 mdbs       496436 Jun  7 10:11 ti.exe*
```

Q. THE "CNTRL/COMMON" DIRECTORY

This directory contains source code common to two or more controller processes. The "Object" subdirectory holds a large quantity of object code for controller processes used by the linker.

```
greg/CNTRL/COMMON:
total 7
drwxr-xr-x  3 mdbs          512 Oct 25  1992 ./
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ../
drwxr-xr-x  2 mdbs          512 Jun 20 13:09 Object/
-rw-r--r--  1 mdbs         3209 Jun  2  1992 tmplsr.c
```

R. THE "CNTRL/CCOM" DIRECTORY

This directory holds the source code needed to create the executable files for the controller's get (CGET) and put (CPUT) processes. The function of the makefile and shell scripts are identical to those discussed for the "BE/BCOM" directory above.

```
greg/CNTRL/CCOM:
total 13
drwxr-xr-x  3 mdbs          512 Jun  2 11:34 ./
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ../
drwxr-xr-x  2 mdbs          512 Jun 20 13:09 Object/
-rw-r--r--  1 mdbs         2059 Jun  2  1992 cget.c
-rw-r--r--  1 mdbs         1225 Jun  2  1992 cput.c
-rw-r--r--  1 mdbs           33 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs          830 Jun 20 13:06 flags.def
-rw-rw-r--  1 mdbs          556 Jun 20 13:09 make_result
-rwxr--r--  1 mdbs          106 Jun  2  1992 mk*
```

S. THE "CNTRL/IIG" DIRECTORY

This directory contains the source code for the insert-information-generator (IIG) processes.

```
greg/CNTRL/IIG:
total 49
drwxr-xr-x  3 mdbs          512 Jun 20 12:25 ./
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ../
drwxr-xr-x  2 mdbs          512 Jun  2 11:34 Object/
-rw-r--r--  1 mdbs        11286 Jun  2  1992 bes.c
-rw-r--r--  1 mdbs          498 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs         4287 Jun  2  1992 didgen.c
-rw-r--r--  1 mdbs          827 Jun  2 11:29 flags.def
-rw-r--r--  1 mdbs        11255 Jun  2  1992 iig.c
-rw-r--r--  1 mdbs          244 Jun  2  1992 iig.dcl
-rw-r--r--  1 mdbs         1295 Jun  2  1992 iig.def
-rw-r--r--  1 mdbs          236 Jun  2  1992 iig.ext
-rw-r--r--  1 mdbs         2931 Jun  2  1992 iigdbl.c
-rw-r--r--  1 mdbs         5936 Jun  2  1992 iigsr.c
-rw-rw-r--  1 mdbs          833 Jun  2 11:34 make_result
-rwxr--r--  1 mdbs          118 Jun  2  1992 mk*
```

T. THE "CNTRL/PP" DIRECTORY

This directory contains the source code for the program supporting the post-processing (PP) process.

```
greg/CNTRL/PP:
total 56
drwxr-xr-x  3 mdbs          512 Jun  2 11:36 ./
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ../
drwxr-xr-x  2 mdbs          512 Jun  2 11:38 Object/
-rw-r--r--  1 mdbs          137 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs          827 Jun  2 11:29 flags.def
-rw-rw-r--  1 mdbs          654 Jun  2 11:37 make_result
```

```
-rwxr--r--  1 mdbs          117 Jun  2  1992 mk*
-rw-r--r--  1 mdbs         8486 Jun  2  1992 pp.c
-rw-r--r--  1 mdbs          157 Jun  2  1992 pp.dcl
-rw-r--r--  1 mdbs         1379 Jun  2  1992 pp.def
-rw-r--r--  1 mdbs          116 Jun  2  1992 pp.ext
-rw-r--r--  1 mdbs         7201 Jun  2  1992 ppby.c
-rw-r--r--  1 mdbs         8195 Jun  2  1992 pprba.c
-rw-r--r--  1 mdbs         5582 Jun  2  1992 ppsr.c
-rw-r--r--  1 mdbs        11447 Jun  2  1992 repmon.c
```

U. THE "CNTRL/REQP" DIRECTORY

This directory holds the source code for the request processing (REQP) process.

```
greg/CNTRL/REQP:
total 119
drwxr-xr-x  3 mdbs          512 Jun  2 11:38 ./
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ../
drwxr-xr-x  2 mdbs         1024 Jun  2 11:40 Object/
-rw-r--r--  1 mdbs        13546 Jun  2  1992 chkptu.c
-rw-r--r--  1 mdbs           35 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs          829 Jun  2 11:30 flags.def
-rw-r--r--  1 mdbs          823 Oct  4  1992 flags.def.nli
-rw-r--r--  1 mdbs         4218 Jun  2  1992 lsrc
-rw-rw-r--  1 mdbs          889 Jun  2 11:40 make_result
-rw-r--r--  1 mdbs         2989 Jun  2  1992 mallocs.c
-rwxr--r--  1 mdbs          107 Jun  2  1992 mk*
-rw-r--r--  1 mdbs        14580 Jun  2  1992 reqcomp.c
-rw-r--r--  1 mdbs        18334 May 12 08:38 reqp.c
-rw-r--r--  1 mdbs          125 Jun  2  1992 reqp.dcl
-rw-r--r--  1 mdbs          677 Jun  2  1992 reqp.def
-rw-r--r--  1 mdbs           91 Jun  2  1992 reqp.ext
-rw-r--r--  1 mdbs        13270 Jun  2  1992 reqpsr.c
-rw-r--r--  1 mdbs        40029 Jun  2  1992 ysrc
```

V. THE "CNTRL/TI" DIRECTORY

This directory, and the many subdirectories beneath it, hold the source code for the test interface (TI) process. The TI process directly interacts with the user by receiving terminal instructions and displaying results. This directory contains the code supporting the kernel data language. The directories beneath this one (under the LangIF subdirectory) contain the code supporting the other model-language interfaces (hierarchical, network, relational, object-oriented, and functional). It is important to observe that all of the source code in this directory and every directory beneath it is a part of the same TI process.

```
greg/CNTRL/TI:
total 207
drwxr-xr-x  4 mdbs         1024 Jun  2 11:40 ./
drwxr-xr-x  9 mdbs          512 Jun  2 11:46 ../
drwxr-xr-x  5 mdbs          512 Mar  5 12:39 LangIF/
drwxr-xr-x  2 mdbs         1024 Jun  7 10:11 Object/
-rw-r--r--  1 mdbs         6555 Mar  5 12:49 dbl.c
-rw-r--r--  1 mdbs          155 Jun  2  1992 dblocal.def
-rw-r--r--  1 mdbs         6052 Jun  2  1992 dblsr.c
-rw-r--r--  1 mdbs          833 Jun  7 09:58 flags.def
-rw-r--r--  1 mdbs         7384 Jun  2  1992 gdb.c
-rw-r--r--  1 mdbs         7149 Jun  2  1992 gsdesc.c
-rw-r--r--  1 mdbs         6214 Jun  2  1992 gsgenrec.c
-rw-r--r--  1 mdbs         4613 Jun  2  1992 gsgmset.c
-rw-r--r--  1 mdbs         4682 Jun  2  1992 gsmodset.c
-rw-r--r--  1 mdbs         2948 Jun  2  1992 gstmpl.c
-rw-r--r--  1 mdbs        31537 Jun  2  1992 intest.c
-rw-rw-r--  1 mdbs         1587 Jun  7 10:11 make_result
-rwxr--r--  1 mdbs          204 Sep 30  1992 mk*
-rw-r--r--  1 mdbs         4729 Apr  2 15:16 ti.c
-rw-r--r--  1 mdbs         9741 Jun  2  1992 tireqs.c
-rw-r--r--  1 mdbs        16120 Jun  2  1992 tireqsubs.c
-rw-r--r--  1 mdbs         9185 Jun  2  1992 tisr.c
-rw-r--r--  1 mdbs        25448 Jun  2  1992 tisubs.c
-rw-r--r--  1 mdbs        20371 Mar  5 13:18 tstint.c
-rw-rw-r--  1 mdbs        21055 Apr 16 15:23 tstint.c.bak
-rw-r--r--  1 mdbs         1049 Jun  2  1992 tstint.dcl
-rw-r--r--  1 mdbs         1441 Jun  2  1992 tstint.def
-rw-r--r--  1 mdbs         1188 Jun  2  1992 tstint.ext
-rw-r--r--  1 mdbs         1961 Jun  2  1992 unixtime.c
```

W. THE "CNTRL/TI/LangIF" DIRECTORY

This directory serves as the parent directory for all of the non-kernel data model/ language interfaces. All of the code supporting these interfaces can be compiled and linked using the makefiles located here. Each model/language interface can also be individually compiled and linked in the corresponding lower level subdirectory. The "include" subdirectory holds header files common to two or more of the non-kernel model/language interface's code. The "lib" subdirectory is where archival copies of the language interface code is stored. The "src" subdirectory leads directly to the code for each model/language interface. These lower level directories are not shown here, but each is logically divided into four sections (kernel controller, kernel formatting system, kernel mapping system, and language interface).

APPENDIX D. BACKEND DIRECTORY AND FILE INFORMATION

The following is an annotated listing of the directories and files making up each backend in the Multibackend Database System. Note that the compiled C executables are normally compiled on the controller and then copied into the proper directories on the backend.

A. THE "/U" ROOT DIRECTORY

The "/u" root directory holds numerous temporary files created, used, and deleted by MDBS. The .pid files are used by MDBS to store process identification numbers generated by the operating system on each run. The .alias file is used by researchers but is not utilized directly by MDBS.

```
drwxrwxr-x 13 mdbs        1024 Feb 27 20:12 ./
drwxr-xr-x 14 root         512 Feb 18  1992 ../
-rw-rw-r--  1 mdbs        1668 Jun 14  1989 .alias
-rw-r--r--  1 mdbs           4 Feb 25 22:56 .bget.exe.pid
-rw-r--r--  1 mdbs           4 Feb 25 22:56 .bput.exe.pid
-rw-r--r--  1 mdbs           4 Feb 25 22:56 .cc.exe.pid
-rwxr-xr-x  1 mdbs         223 Jan 14  1989 .cshrc*
-rw-r--r--  1 mdbs           4 Feb 25 22:56 .dio.exe.pid
-rw-r--r--  1 mdbs           4 Feb 25 22:56 .dirman.exe.pid
-rw-r--r--  1 mdbs           4 Feb 25 22:56 .recp.exe.pid
-rw-r--r--  1 mdbs           2 Feb  4 14:35 .rhosts
-rw-rw-r--  1 mdbs        1189 Feb  4 14:35 .rhosts.bak
drwxrwxr-x  2 mdbs         512 Feb 25 22:56 Sockets/
drwxrwxr-x  2 mdbs         512 Feb 12 11:58 UserFiles/
drwxrwxr-x  2 mdbs         512 Nov  5 13:50 be.greg/
drwxrwxr-x  2 mdbs         512 Feb  4 19:49 bin/
```

B. THE "Sockets" DIRECTORY

The "sockets" directory (under the root directory) contains the six sockets used for interprocess-communication on each backend. These sockets are deleted and created anew for each run of MDBS.

```
Sockets:
total 2
drwxrwxr-x  2 mdbs         512 Feb 25 22:56 ./
drwxrwxr-x 13 mdbs        1024 Feb 27 20:12 ../
```

```
srwxrwxrwx  1 mdbs              0 Feb 25 22:56 CC=
srwxrwxrwx  1 mdbs              0 Feb 25 22:56 DIO=
srwxrwxrwx  1 mdbs              0 Feb 25 22:56 DM=
srwxrwxrwx  1 mdbs              0 Feb 25 22:56 G_PCLB=
srwxrwxrwx  1 mdbs              0 Feb 25 22:56 P_PCLB=
srwxrwxrwx  1 mdbs              0 Feb 25 22:56 RECP=
```

C. THE 'UserFiles" DIRECTORY

This directory, located directly under the root directory, holds the descriptor (*.d) and template (*.t) files for each database (*e.g.* SALES) used by MDBS. Both of these files must be present or the database will not run. The below listing indicates that this backend is prepared to support three databases (*i.e.* COURSE, SALES, and SQD). For details on the composition of the descriptor and template files see [Bourgeois, 1993].

```
UserFiles:
total 12
drwxrwxr-x  2 mdbs            512 Feb 12 11:58 ./
drwxrwxr-x 13 mdbs           1024 Feb 27 20:12 ../
-rw-r--r--  1 mdbs             51 May 21  1992 COURSE.d
-rw-r--r--  1 mdbs            126 May 21  1992 COURSE.t
-rw-rw-r--  1 mdbs            140 Oct 19  1988 SALES.d
-rw-rw-r--  1 mdbs            121 Oct 19  1988 SALES.t
-rw-rw-r--  1 mdbs             43 Feb 12 10:50 SQD.d
-rw-rw-r--  1 mdbs            140 Feb 12 10:50 SQD.t
```

D. THE "be.version" DIRECTORY

This directory, located under the root directory, is the top-level directory for each different version of MDBS software on each backend. . In this case, the working version is "greg" (hence "be.greg"). This directory contains the executable files (*.exe) for the six backend processes. It also holds the trace files (*.tr). The trace files are text files output by the processes of the same names. The run.be script is used by the controller to start the six backend processes.

```
be.greg:
total 2477
drwxrwxr-x  2 mdbs            512 Nov  5 13:50 ./
drwxrwxr-x 13 mdbs           1024 Feb 27 20:12 ../
-rwxr-xr-x  1 mdbs          83649 Nov  5 13:23 bget.exe*
-rw-r--r--  1 mdbs              0 Feb 25 22:56 bget.tr
-rwxr-xr-x  1 mdbs          83649 Nov  5 13:23 bput.exe*
```

```
-rw-r--r--  1 mdbs           60 Feb 25 22:58 bput.tr
-rwxr-xr-x  1 mdbs       246357 Nov  5 13:23 cc.exe*
-rw-r--r--  1 mdbs           37 Feb 25 22:58 cc.tr
-rwxr-xr-x  1 mdbs        89208 Nov  5 13:23 dio.exe*
-rw-r--r--  1 mdbs          465 Feb 25 22:58 dio.tr
-rwxr-xr-x  1 mdbs       375581 Nov  5 13:23 dirman.exe*
-rw-r--r--  1 mdbs          101 Feb 25 22:58 dirman.tr
-rw-r--r--  1 mdbs           19 Nov  4 17:10 exe.awk
-rwxr-xr-x  1 mdbs       335559 Nov  5 13:50 recp.exe*
-rw-r--r--  1 mdbs           37 Feb 25 22:58 recp.tr
-rwxr--r--  1 mdbs          431 Nov  5 13:37 run.be*
-rw-r--r--  1 mdbs          160 Nov  4 17:10 stop.exe
```

E. THE "bin" DIRECTORY

Located under the greg.be directory, the bin directory holds the utility files used by the backend. The most important of these is the zero command (zero*) which is used to initialize the meta and data disks before each run. The other utilities present here are leftover from earlier versions of MDBS. They are no longer required to run MDBS, but have been left here because they can sometimes be useful. The stop command (stop.cmd) may be used to stop MDBS processes running on this backend. The .list.stop and exe.awk files work with the stop command. The cpcount.c file copies a user-specified amount (in bytes) of an existing file to a new file specified by the user. The cpydisks script was once used to redistribute backend code.

```
bin:
total 39
drwxrwxr-x  2 mdbs          512 Feb  4 19:49 ./
drwxrwxr-x 13 mdbs         1024 Feb 27 20:12 ../
-rw-r--r--  1 mdbs            0 Jun 14  1989 .list.stop
-rw-rw-r--  1 mdbs         1610 Jan 14  1989 cpcount.c
-rwxrwxr-x  1 mdbs           84 Jan 20  1989 cpydisks*
-rw-r--r--  1 mdbs           19 Nov  5 12:52 exe.awk
-rwxr-xr-x  1 mdbs          373 Feb  5 13:49 stop.cmd*
-rw-r--r--  1 mdbs            0 Feb  4 18:50 stop.trace
-rwxr--r--  1 mdbs        26439 Nov  5 12:52 zero*
```

APPENDIX E. DEMONSTRATION DATABASE

The new AMMO (ABDL) database created for the demonstration of MDBS on the new hardware and software platform is listed below.

A. THE DESCRIPTOR FILE(AMMO.d)

```
AMMO
TEMP b s
! INFO
! COUNT
@
DODIC a s
A001 J999
K001 Z999
@
NOMEN a s
A G
H R
S Z
@
QTY a i
1 100
100 1000
@
$
```

B. THE TEMPLATE FILE (AMMO.t)

```
AMMO
2
3
INFO
TEMP s
DODIC s
NOMEN s
3
COUNT
TEMP s
```

```
DODIC s
QTY i
```

C. THE RECORD FILE (AMMO.r)

```
AMMO
@
INFO
D680 Projo
D681 Projo
N232 Fuze
N340 Fuze
@
COUNT
D680 200
D681 150
N232 180
N340 170
$
```

D. THE QUERY FILE (AMMOreq1)

```
[RETRIEVE(TEMP=INFO) (DODIC,NOMEN)BY DODIC]%
[RETRIEVE(TEMP=COUNT) (DODIC,QTY)BY DODIC]%
[INSERT(<TEMP,INFO>,<DODIC,M130>,<NOMEN,PROP>)]%
[DELETE((TEMP=INFO) and (DODIC=M130))]%
```

APPENDIX F. NEW MDBS FUNCTIONS

During this porting project, numerous adjustments and modifications were made to the existing code. A few existing functions were almost completely rewritten. New constants were also added as needed. Consistent with the goals outlined in Chapter 1, though, only two wholly new functions with their associated calls were added to MDBS.

A. THE "host_name_integer" FUNCTION

This function, located in the ack.c source file, receives a hostname (which may be a member of the host_names array) and returns only the number portion as an integer. This integer is used by other functions to uniquely identify the workstation. This function is called from many locations within ack.c.

```
int host_name_integer(host_name)
/* this function is passed an element of the host_names array
(e.g."dbl1") - it returns the number part as an integer. This
routine replaces the old way of picking the number part of the host
name.*/

   char host_name[];

{
   char temp[host_name_len + 1];
   int temp_index = 0;
   int i;

   for(i=0;i<host_name_len;i++){
      if(isdigit(host_name[i]) != 0){
         temp[temp_index] = host_name[i];
         temp_index++;
      }
   }
   temp[temp_index] = '\0';

   return(atoi(temp));
} /* end host_name_integer() */
```

B. THE "init_meta_NATT" FUNCTION

This function, located in the meta.c source file, uses the global definitions of first_record_cylinder and first_record_track (meta.def) to load initial values to the Next-

Available-Track-Table (NATT) on the meta-data disk. This function is called from the code in the dirman.c source file.

```
init_meta_NATT()
   /* Store record disk's starting cylinder/track values to NATT */
{
   unsigned short fir_rec_cyl = first_record_cylinder;
   unsigned char fir_rec_trk = first_record_track;
   long lseek();

#ifdef EnExFlag
   printf("Enter init_meta_NATT\n");
   fflush(stdout);
#endif

   /* seek to NATT area of meta disk */
   if (lseek(metafptr, (long) NATT_OFFSET, 0) != NATT_OFFSET)
      SysError(8, "1 init_meta_NATT");

   /* write initial valuse */
   if (write(metafptr,&fir_rec_cyl,sizeof(fir_rec_cyl)) < 0)
      SysError(12, "2 init_meta_NATT");
   if (write(metafptr,&fir_rec_trk,sizeof(fir_rec_trk)) < 0)
      SysError(12, "3 init_meta_NATT");

#ifdef EnExFlag
   printf("Exit init_meta_NATT\n");
   fflush(stdout);
#endif
} /* end init_meta_NATT */
```

LIST OF REFERENCES

Bourgeois, Paul A., "The Instrumentation of the Multimodel and Multilingual User Interface," M. S. Thesis, Naval Postgraduate School, Monterey, California, March, 1993.

Boyne, Richard D. and Demurjian, Steven A. and Hsiao, David K. and Kerr, Douglas S. and Orooji, Ali, "The Implementation of a Multi-Backend Database Syetem (MDBS): Part III," Technical Report, Naval Postgraduate School, Monterey, California, March, 1983.

De Witt, David J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, Vol. C-28, No. 6, June, 1979, pp. 395 - 406.

Elmasri, Ramez and Shamkant B. Navathe. Fundamentals of Database Systems. New York, The Benjamin/Cummings Publishing Company, Inc., 1989.

Hall, James E., "Performance Evaluations of a Parallel and Expandable Database Computer - The Multi-Backend Database Computer," M. S. Thesis, Naval Postgraguate School, Monterey, California, June, 1989.

Hammond, Greg Alan, "The Instrumentation of a Parallel, Distributed Database Operation, Retrieve-Common, for Merging two large sets of records," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1992. (H17526)

He, Xingui and Higashida, Masanobu and Hsiao, David K. and Kerr, Douglas S. and Orooji, Ali and Shi, Zong-Zhi and Strawser, Paula, "The Implementation of a Multi-Backend Database System (MDBS): Parts II and III," Technical Report, Naval Postgraduate School, Monterey, California, July 1982.

Hsiao, David K., "A Parallel, Scalable, Microprocessor-Based Database Computer for Performance Gains and Capacity Growth," IEEE Micro, December 1991, pp. 44-60.

Hsiao, David K., "Federated Databases and Systems: Part I - A Tutorial on Their Data Sharing," Very Large Database (VLDB) Journal, vol 1, no 1, 1992, pp. 127-179.

Hsiao, David K., "Federated Databases and Systems: Part II - A Tutorial on Their Resource Consolidation," Very Large Database (VLDB) Journal, vol 1, no 2, 1992, pp. 285-310.

Hsiao, David K. and Kamel, Magdi N., "Heterogeneous Databases: Proliferations, Issues, and Solutions," IEEE Transactions of Knowledge and Data Engineering, vol 1, no 1, March 1989, pp. 45-62.

Kloepping, Gary R. and Mack, John F. Mack, "The Design and Implementation of a Relational Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985. (K587163)

Leffler, Sam and Fabry, Robert S. and Joy, William N. and Lapsley, Phil, "An Advanced 4.3BSD Interprocess Communication Tutorial," Integrated Solutions UNIX Programmers Supplementary Documents, July, 1987.

Little, Craig W., "The Design and Implementation of Pedagogical Software for the Multi-Backend/Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1987. (L692)

Meeks, Andrew P., "The Instrumentation of the Multibackend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1993.

Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language (Second Edition). Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1988.

Que Corporation, Using UNIX, Carmel, Indiana, Que Corporation, 1990.

Rieken, Bill and Weiman, Lyle, Adventures in UNIX Network Applications Programming. New York, New York, John Wiley and Sons, Inc., 1992.

Rochkind, Marc J., Advanced UNIX Programming. Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1985.

Rosch, Winn L., The Winn Rosch Hardware Bible, New York, New York, Simon & Schuster, Inc., 1989.

Rosen, Kenneth H. and Rosinski, Richard R. and Farber, James M., Unix, System V Release 4, An Introduction. Berkeley, California, Osborne McGraw-Hill, 1990.

Stevens, Richard W., Advanced Programming in the UNIX Environment. Reading, Massachusetts, Addison-Wesley Publishing Company, Inc., 1992.

United States House of Representatives, "DoD Automated Information Systems Experience Runaway Costs and Years of Schedule Delays While Providing Little Capability," Report 101-382, November 1989.

Wong, Albert, "Toward Highly Portable Database Systems: Issues and Solutions," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1986. (W755)

Zawis, John A., "Accessing Hierarchical Databases via SQL Transactions in a Multi-Modal Database System," M. S. Thesis, Naval Postgraguate School, Monterey, California, December, 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	2
4.	Ms. Doris Mlezko Code P22305 NAWCWPNS Point Mugu, CA 93042-5001	2
5.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
6.	Professor David K. Hsiao, Code CS/Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
7.	Ronald J. Roland 500 Sloat Avenue Monterey, CA 93940	1
8.	Major Stanley H. Watkins, USMC 6701 Abbey Road Bartlesville, OK 74006	1

END